New Operations for Display Space Management and Window Management

Technical Report GIT-GVU-02-18
August 2002

Dugald Ralph Hutchings *and* **John Stasko**
College of Computing / GVU Center
Georgia Institute of Technology
Atlanta, GA 30332 USA
{hutch, stasko}@cc.gatech.edu

Abstract

We present a set of new operations for managing screen real estate that allow windows to acquire more desktop space. This set of operations obeys the following guidelines: (1) the visible information contents of each window are preserved (i.e., operations never result in covering already-exposed window contents), (2) operation invocation requires only simple user action, and (3) windows grow and move in a natural and easily understandable manner, mimicking the interactions of colliding physical objects. We call the main operations *expand* and *shove*. Expand and shove represent two endpoints on a space-acquisition scale, and we give other possible operations called *jostle* and *ram* that fall between these two points. Additional concepts of *undo* (to allow windows to revert to earlier sizes and positions) and *relevant regions* (to allow more tightly controlled window information regions to be indicated) are also introduced. To theoretically support the methods used by the operations, we present a classification of the possible ways that pairs of windows can initially overlap and subsequently interact during the operations.

Keywords

window, window management, desktop, physics, physically-based, information preservation, relevant regions, animation, space management

1. INTRODUCTION

Several aspects of computer science have experienced explosions in the last five or ten years. The most well-known and perhaps oft-quoted idea is the "Information Explosion," i.e., that the amount of human-readable and -accessible information that computers can easily generate and process has grown in an exponential fashion. This increase in both type and quantity of computer-delivered information has prompted people to use computer technology in an ever-increasing variety of ways. People often use devices with small displays (for example PDAs and wireless telephones) to display and interact with personally-oriented, small-scale, or frequently-accessed information. Researchers have already explored ways that very large displays (whether large in size, resolution, or both) can support large-scale or information-intensive tasks. Both small and large display devices can help users to complete several disparate tasks simultaneously; advances in chip speed and monitor size and quality have similarly allowed people using a traditional "desktop setup" to work on many different things in a small window of time.

With the Information Explosion acting as a catalyst in all of these other areas, many challenging questions arise. One particular question is "How does one easily manage the simultaneous display of tasks and information?" Traditionally, tasks are divided into distinct spaces, such as windows on a desktop. When the desktop metaphor and its corresponding notion of window-based interaction was introduced approximately twenty-five years ago, four fundamental operations were available for interacting with and managing windows. The functions are *add* (create a new window), *delete* (remove an existing window), *reposition* (move an existing window to a new location), and *resize* (change the extent of an existing window in one or both dimensions). These fundamental operations have remained largely unchanged, although different windowing systems provide other fundamental functions (for example, Microsoft Windows has maximize and minimize functions, but both of these are essentially specialized resize functions). Other windowing systems have additional functions for desktops that allow windows to overlap (for example X has the functions *raise* and *lower* for placing a window at different levels in the overlap scheme).

The fundamental operations have maintained quite a high degree of stability, but we contend that their effectiveness in managing display space has decreased in light of the events described above. In particular, as people have generally started to work on a large number of very different tasks simultaneously, yielding a desktop with a large number of windows in view at one time, windowing operations that completely ignore other tasks/windows contribute to increasing amounts of required window management time (see for example [8] for an early account). Previous work in allowing users to manage all of their windows has focused on different external systems or approaches commonly called *window managers*. There are quite a number of window managers (see for example [6]), each of which consider windows in isolation or possibly as groups. However, window managers often (or perhaps always) fail to specifically account for empty space on a display, especially in relation to users whose needs and tasks constantly vary. The work that we present attempts to account for such users. First though, we present some related work and analyze this work in light of such users.

2. RELATED WORK

We will analyze a number of ideas from literature on window and space management in this section. We first view some recent management systems. A discussion of other topics and ideas follows.

2.1 Recent Systems

Elastic Windows [10, 11] acknowledges the need for advanced windowing operations for users with many tasks. The system's philosophy involves a role-based, hierarchically-defined, space-filling tiling paradigm. The desktop acts as the root window that contains other windows, which in turn contain other windows, and so on. The higher levels of windows apply to certain characteristics of the user's current task(s). For example, consider a college professor, who at the first level is a class instructor, and at the next level is an instructor of a particular class (thus giving us the hierarchy and role components) [10]. The system arranges windows in such a way that first-level windows do not overlap and consume the entire space of the desktop, the second-level windows do not overlap and consume the entire space of the first level window in which they reside, and so on (yielding space-filling tiles). When a window *W* in level *n* is resized, all siblings of *W*, as well as all children of *W*, grow or shrink accordingly, which subsequently implies potential changes in every window. Thus, we have potentially global desktop effects with only one window movement.

While some advantages were shown in [10], there are also some disadvantages with respect to aiding users who desire to complete many different tasks simultaneously or in a small time span. For example, since enlarging one window will shrink many other windows, original contents of other windows (tasks) are hidden or altered beyond recognition. This obviously makes it difficult for a user to multitask across roles (for example, write a paper for a class and maintain instant messaging discussion with a friend), and disallows good *glancing-only* applications (for example, a clock or a weather conditions report). Furthermore, space-filling tiling is the only option, which limits the number of windows that can effectively display information simultaneously. Further discussion of tiling is found in other sections of this paper.

The Scheme Constraints Window Manager (SCWM) [1], is obviously rooted in a constraint-based system (see [5] for an example of an older system). However, users need not be familiar with the abstract notion of constraints in order to use the system; the constraints built are masked by allowing users to specify certain types of *relationships* specific to the notion of a window. For example, two windows can be defined to be "always adjacent" so that dragging one also drags the other. These relationships are available to the user through a palette of icons depicting the different types of relationships. Additionally, operations can be recorded and combined to define and perform new relationships.

While this is an innovative and interesting system, there are two potential pitfalls in a relationship-based format. The first is user forgetfulness: if a user makes a relationship and subsequently forgets that the relationship exists, or makes a relationship that inadvertently breaks another relationship, the consequences of a single action could be confusing and overwhelming. The second, and perhaps most relevant to our work, relates to effective and simple multitasking. While defining a routine that accounts for all other windows and the edges of a desktop is possible, such a routine is difficult to delineate in a relationship-based system without defining, and subsequently maintaining, a large number of relationships. Furthermore, relationships must be defined per session (machine and window instance) and since user desktops can change many times per session [8], this may be a time-consuming process. Our work seeks to allow users to exert only a small amount of effort to window relationships and focus on the use of space in general, non-relationship-based ways. Note that SCWM also has the problems inherent in multi-way constraint systems, such as unpredictability [14].

Components of the Flatland system [15] present an interesting way for windows to interact with each other while supporting multitasking. Each *segment* (which is essentially equivalent to a window) moved by the user can "bump" into other segments, moving inactive segments out of the way, but never covering the information contents (Flatland is a tiling-based system). Moreover, inactive segments can shrink to allow more space to an active segment without changing layout of information in the segment. We will discuss how we also use notions of physically-based interactions, but applied to the more general notion of an overlapping desktop space.

2.2 Other topics

Beaudouin-Lafon [3] recently contributed a number of techniques for window interaction on the overlapping desktop. His "tabbed windows" are special windows that contain arbitrarily many standard windows and assign a tab to each. Desktop space is saved by this grouping at the cost of only allowing one window per group to be visible at one time. He also introduces paper-based notions of "rotation," which allows users to slightly rotate a window, and "peeling back," which allows a user to view and interact with windows that exist underneath the peeled-back window. These techniques allow users to more abstractly and more efficiently use space and highlights a possibility of understandable physically-based operations.

A number of authors have explored adaptive window management techniques (see for example [7, 12, 18]). The core idea of these systems is that the way a user manages and interacts with windows can be analyzed. This analysis can be used to automatically perform window management without explicit user action. Common to the adaptive approach is to shrink, or completely remove, windows deemed to be "inactive." But this is a dangerous notion, especially since "not frequently interacted with" does not mean "not used by the user" (also noted in [13]). Many windows serve a browsing or glancing purpose: the information is seen by the user but the user never operates upon the window. Even in the case of shrinking, the information layout in the window may be completely altered, thus removing its glancing capability (although some monitoring tasks have been aided by windows that automatically shrink in relationship to other windows; see for example [2, 17, 20]). A much stronger (and likely user-based) model of window or task management would be in order for these techniques to be acceptable.

Although not necessarily related to window management, the Data Mountain system [16] offers some interesting ideas with respect to treating objects in groups and utilizing empty space to avoid occlusion. The system of Robertson et. al. was specifically aimed at allowing users to remember bookmarked web pages by providing a thumbnail image of the page and a three-dimensional space in which to place the image. Images could be arranged in groups (thus exploiting spatial cognition), and when an image was being placed, groups would move to allow the new image to take space in the group without completely obscuring any one member of the group. This allowed the users to treat overlapped images as groups and manage the groups in predictable ways.

A final piece of work to mention is related to the empty space on the desktop. Bell and Feiner [4] propose a system that efficiently calculates the largest empty-space rectangles available on a desktop that allows overlap. They use this to support "non-overlapping dragging" where stationary windows that are "hit" by a dragged window are moved to the nearest available empty space. While our work does not specifically incorporate their representation, their notion that empty space calculation is important has motivated the ideas that we will discuss. One potential aspect of our future work may incorporate their representation system.

3. OPERATIONS AND CONCEPTS

In the previous section, we uncovered a number of disadvantages of current and past work in window and space management, and now we present work which attempts to overcome the disadvantages. One goal is aiding any users who desire to be able to *simply* maintain several different tasks simultaneously. We thus propose a new group of *operations*, and not a manager, that attempt to aid users in administering their screen real estate by *maintaining the visible portions of other windows*. To do so, and since the fundamental operations of move and resize are so entrenched in the user population, these new operations exploit the reposition and resize operations. Similar to the concept of Flatland [15], our contention is that attribution of physical properties of solid objects to windows will aid in comfortably repositioning many windows simultaneously. This is the way that a real desktop works; if a person needs to clear some space for one task but still needs to access and view information from other tasks, the other tasks can be pushed out of the way, to the sides and corners of the desk. The operations that we propose try to mimic this notion.

As mentioned, however, Flatland [15] only allows for a tiling of windows, whereas we desire to support users who use overlapping windows on their desktops (similar to the way that papers can be piled on top of each other). This significantly confounds the idea of "bumping" windows on the desktop, since we are now forced to think about interaction in *three* dimensions (or, as some have called it, "two and a half") rather than *two* dimensions. For example, consider Figures 1 and 2 below. If a user grows the shaded window in Figure 1, other windows are bumped in a rather obvious way. However, consider the window *G* in Figure 2. Remember, we want to maintain the visible portions of *X*, *Y*, and *Z*. But if *G* grows upward, then *X* must move upward (otherwise *G* will cover more of *X*), and if *X* moves upward, then *Z* must move upward (otherwise *X* will cover a different part of *Z*), and if *Z* moves upward, then *Y* must move upward (otherwise *Y* will cover a different part of *Z*). But if *Y* moves upward, then *G* will cover more of *Y*, because the bottom edge of *G* remains stationary as *G* grows upward. The same situation occurs when *G* attempts to move downward. Indeed, the only available growth directions for *G* are rightward (since no windows are in the way) and leftward (thus "pushing" *X*, *Y*, and *Z* simultaneously).

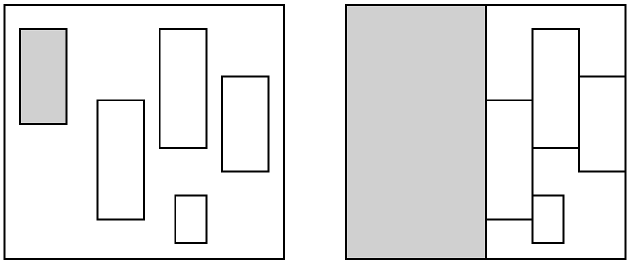


Figure 1: The shaded window bumps other windows on the desktop in an obvious way.

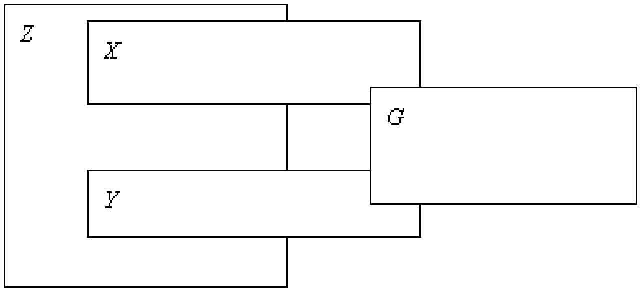


Figure 2: Where do *X*, *Y*, and *Z* move as *G* grows?

Figures 3 and 4 highlight different issues. In Figure 3, *W* starts to grow, thus pushing *M* to the top and to the left simultaneously. The issue here is what to do with the window *Z*, which initially overlaps neither of *W* nor *M*. *Z* could act as a moveable object and be bumped out of the way by *M*, or *Z* could act as an immovable object, preventing *M* from bumping *Z* out of the way. In this figure, *Z* is immovable, like a wall. Note how this allows *M* to "sneak around the corner" of *Z*, so that *W* finally takes a little more space than when *M* comes into contact with *Z*. In Figure 4, *W* grows, thereby pushing *M* up and to the left. But, the question is what to do with *Z*: does it also move up and to the left (being "attached" to *M*) or only upward, since when *M* moves left, **more** of *Z* is revealed (being moved by *M* and *W* only when necessary). In this figure, *Z* assumes the latter role. Note how *W* later bumps into *Z*, thereby moving it upward since *Z* was originally mobile (and which is thus slightly different than the situation in Figure 3; especially compare Figure 3a with Figure 4c).

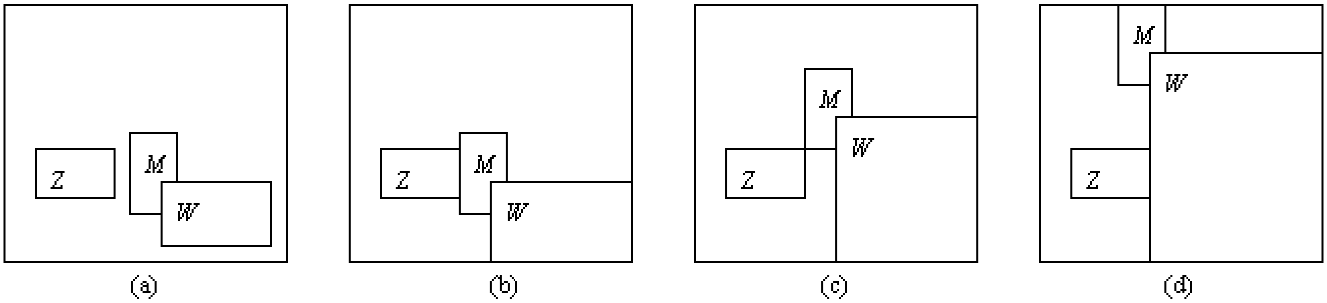


Figure 3: *W* begins growth left in (a), and then (b) stops when *M* becomes adjacent to *Z*. But (c) there is enough room for *M* to "sneak around the corner" of *Z*, and so (d) *W* resumes movement in the left direction.

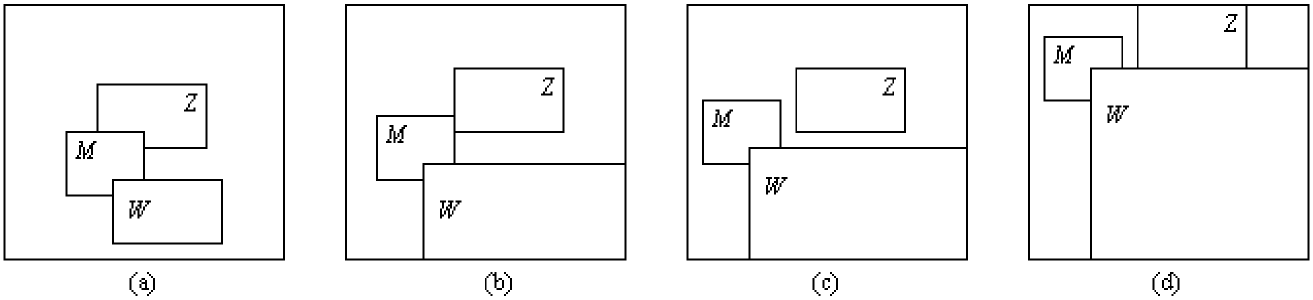


Figure 4: (a) *W* begins growth top and left, but (b, c) *Z* stops when *M* no longer intersects *N*. But (d) *W* then becomes adjacent to *Z*, and so *Z* resumes its movement toward the top.

We address these types of questions with the introduction and explanation of our operations, called *expand*, *jostle*, *ram*, and *shove*. Expand and shove constitute two endpoints along a space-acquisition scale, with shove representing the operation that takes as much space as possible. The jostle and ram functions represent intermediate locations on this scale. Complementing all of these operations is an *undo* operation. We also discuss the importance of animation in using these operations. First, however, we discuss some core ideas behind these operations and then define some terms that are used throughout this paper.

3.1 Discussion

As stated, we follow three guidelines for our proposed operations. One guideline, which we have also already analyzed, is aiding users in administering their screen real estate by *maintaining the visible portions of other windows* during the operations, i.e., that our operations never result in the covering of an exposed portion of a window. Another guideline is that users should be able to *simply* maintain several different tasks simultaneously. This idea is inherent in the operation-based format of our work. Consider the maximize operation available on many systems (such as Microsoft Windows or Sun's CDE). One simple click of the mouse on a button in the title bar of the window allows the window to take all available screen space. Although the ideas that we have already discussed involve several windows moving simultaneously, our current prototype provides these operations through a simple click of the mouse button (although on a menu embedded in the window border, not a button in a fixed location).

To move windows simultaneously, we employ a notion of physically-based operations. But what does it mean to be "physically-based?" There are several possible connotations with this concept, especially since many different physical objects may move differently in relation to other physical objects. For example, consider two child's blocks wrapped in Velcro. When the two blocks come into contact with each other, they "attach," and subsequently move as one block-shaped object. Now consider the blocks wrapped in a slippery substance, such as grease. The block that is being pushed by the other block will only move in a particular direction when the friction coefficient is overcome by the amount and direction of force exerted by the other block. Clearly, both situations are "physically-based," but it is unclear which one is the "right" approach for moving windows.

Since our other guideline for interaction involves the maintenance of the visible portions of every window, we have adopted the following approach. A growing window can move another window in only two possible directions at a time. We move the other window in either direction if and only if *not* doing so would result in covering an exposed portion of that window. In other words, treat each direction independently, and only move when a visible region of a window is about to be obscured. For example, consider Figure 4 again. *W* begins to grow, thus moving *M* toward the top and left simultaneously. But when *M* moves, *Z* too has to move. However, since moving *Z* to the left results in *more* of *Z* being exposed, we only move *Z* to the top.

3.2 Definitions

We will use a number of common terms throughout the remainder of this paper. To avoid any confusion, we give precise definitions of these terms here. When we talk about a *window*, we mean a rectangular object occupying space on a pixel-based display. The outermost pixels of a window are the *border* (or *sides*), covering the four basic directions *left*, *top*, *right*, and *bottom* (which we commonly refer to as L, T, R, and B, respectively). In the case of a desktop that allows for overlap, two windows are said to *intersect* or *overlap* if the windows share at least one common pixel. Two windows are *adjacent* if a section of each of their borders touch (so that there are no pixels in between the two windows), but the two windows do not intersect. In view (b) of Figure 5E, the shaded window intersects the window beneath it, and the shaded window is adjacent to the window to its right. Any desktop region that contains no windows is *empty space*.

3.3 Expand

The first concept is *expand*. Here, one selects a window for growth in all directions. The selected window *S* grows independently (but simultaneously) in each direction *d* until *S* becomes adjacent to some other window in *d* or reaches the *d*-edge of the screen. If some part of the *d*-border of *S* intersects another window, then *S* never grows in

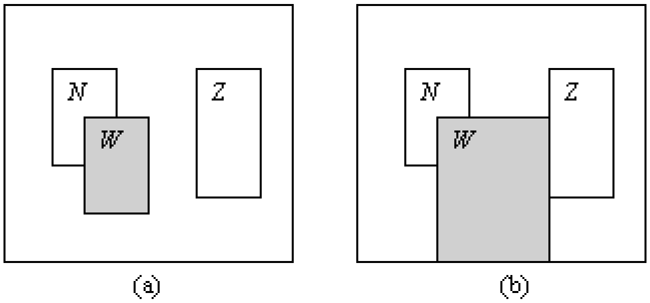


Figure 5E: (a) A simple desktop. (b) The shaded window *W* is expanded. Growth leftward and upward is prevented by the neighbor *N*. Growth rightward and downward are stopped by *Z* and edge of the desktop, respectively.

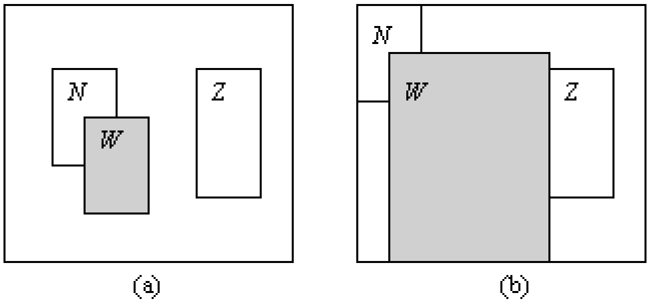


Figure 5J: (a) A simple desktop. (b) The user performs a jostle on *W*. *W* grows leftward and upward, "jostling" its neighbor *N* until *N* hits the edge of the desktop.

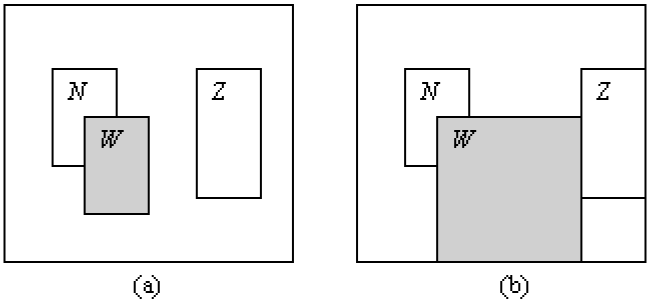


Figure 5R: (a) A simple desktop. (b) *W* has the ram operation executed. *W* grows rightward and downward as in 5E, but also moves *Z* to the right when *W* becomes adjacent to *Z*.

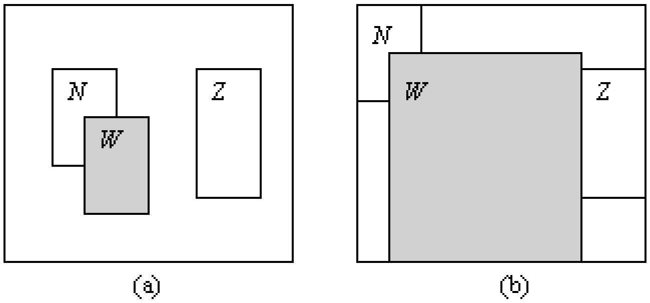


Figure 5S: (a) A simple desktop. (b) The user performs a shove on *W*. *W* pushes *N* as in 5J, and *W* shoves *Z* rightward until *Z* hits the edge of the screen as in 5R.

the *d* direction. Compare Figure 5E views (a) and (b). The shaded window *W* has been expanded in (b). This is a very simple action, allowing one to allocate unused space quickly and to prevent supplanting or additional covering of any other window on the desktop. This operation could be useful, for example, when a small amount of work needs to be completed in one window, leaving all other window positions constant for the next task switch.

3.4 Jostle

A possible disadvantage of expand is that the space allocated to the selected window is insufficient. We thus take a further step in allocating empty space through the function *jostle*, which dictates the repositioning (but not resizing) of additional nearby windows on the desktop. In this case, the selected window *S* "jostles" for position. An example of the utility of such an operation is a desktop that contains monitor windows on the borders of the display area. The user may want to have highly predictable (i.e. fixed) locations for the monitor tool windows (such as mail or CPU load tools), but any other windows on the desktop are candidates for new positions.

Before we precisely discuss jostle, let us define the *neighborhood* of a window. Consider the desktop as a graph, where there is one node per window and an edge between nodes if and only if the represented windows intersect. Then all nodes that have a path to the node for the jostling window *S* constitute the neighborhood of *S* (mathematically speaking, we form the transitive closure of the overlap relation). Two windows need not overlap to be in the same neighborhood (for example, in Figure 2, *G* and *Z* are in the same neighborhood). In fact, all windows of Figure 2 constitute a neighborhood.

When a window *S* jostles, the effect is similar to expand, except each window in the neighborhood of *S* moves in the "appropriate directions" while *S* grows in order to maintain its initially visible portion (so if a neighbor *N* is located mostly to the bottom of the jostling window *S*, then *N* moves toward the bottom). Once a window moving in a direction *d* has reached the *d*-edge of the screen or becomes adjacent with a non-moving window, growth in *d* ceases. For example, compare Figure 5J views (a) and (b). The shaded window *W* "jostles" its neighbor *N* to the top and left as *W* grows, but again ceases growth rightward when *W* strikes *Z*. The main idea is that windows nearby are jostled by *W* to let *W* gain more space but to maintain the visible contents of all windows.

3.5 Ram

The jostle operation seems well suited for a situation with important fixed windows near the edge of the display space. Alternatively, it could be the case that nearby windows should remain stationary (for example, in a documentation window that is supporting a window for programming) while far away windows are inconsequential to the current task. Thus we also posit the operation of *ram* which does not disturb windows in the neighborhood but does supplant windows not in the neighborhood. Figure 5R demonstrates a simple example of ram. The key here is that when an adjacency arises, the window that was previously not moving also begins to move. *W* becomes adjacent to *Z*, and then rams *Z* to the edge of the desktop. Ramming can also have a "domino effect" for desktop windows, by successively ramming additional windows. See Figure 1 for an example, where the shaded window acts as a rammer.

3.6 Shove

Shoving is, in a sense, the union of a ram and a jostle; all windows are candidates for movement. The windows in the neighborhood are moved as with a jostle. However, when non-neighborhood windows are encountered, those windows move as with a ram. For example, compare Figure 5S views (a) and (b). The shaded window *W* shoves *N* to the top and left as in jostle, but also shoves *Z* to the right once *W* strikes *Z*, as in ram. We see shoving as useful whenever a window will be the main focus for a long time, thus justifying a large amount of screen space for the work session.

3.7 Undo

As with any windowing operation that automatically changes more than simply the selected window, the resulting layout could have undesirable characteristics that were unforeseen by the user. We therefore have incorporated a notion of "undo" for the windowing operations. For any uninterrupted series of moving, resizing, or any of our proposed operations, the user may effortlessly recover back through the operations by repeatedly issuing an undo command. By uninterrupted, we mean that there are no window creations or deletions. Undoing a creation would delete the window, which is a dangerous operation, and undoing a deletion of a window is highly problematic, since application information would have to somehow be recovered. Future work may aim to allow undo to operate despite window creation and deletion. Note that undo works at the operation level, so, for example, if a shove operation results in *n* windows changing position, undoing the shove would return those *n* windows to the original positions.

As an example, suppose that several windows have been arranged by a user to complete a specific task. Once done, the user might move a couple of windows, then issue a shove command to the window of interest. While working on the new task, the user remembers that she forgot to complete one piece of the old task. Now, rather than rearranging each window individually, the user may simply issue undo to "unshove" the desktop, and then undo twice more to "unmove" the first two moved windows. This will return the user to the configuration enjoyed for the old task.

3.8 Animation

Expand, jostle, ram, and shove should be fairly easy to understand at a conceptual level, since they are based on physical movement of objects. One is able to understand the transition between views (a) and (b) in Figure 5 because the "before" and "after" states are available simultaneously. Clearly this is not the case on the standard desktop, and therefore we felt it crucial to add the element of animation to our system. Animation allows the user to be able to actually see the movement of the windows on the screen, which we hope will aid in learning and understanding the operations. Our personal reaction to the operations in our very first prototype, which was unanimated, was that the operations were somewhat disorienting and difficult to understand. Once animation was added, however, the operations became much more easy to understand.

4. IMPLEMENTATION FRAMEWORK OVERVIEW

The implementation of the expand, jostle, ram, and shove algorithms consists of two fundamental parts. The first is what we call a *processing* step. Before we can begin to grow the selected window (and move other windows for the shoving operation), we must determine how the other windows on the desktop relate to the growing window and to each other. This informs us both of the directions in which the selected window can grow, and how every other window must move. Once the processing step is complete, we have the appropriate information for actual growth of the selected window and movement of all other windows. This second step of actual manipulation is called the *update* step. The basic idea is to determine the initial directions for growth and then cycle between two subphases: grow in the available directions, and then check the result to ensure that the directions are still available (stopping growth and movement in that direction if not).

The following two sections each cover one step of the algorithm. Remember, we discussed in the previous section that overlapping desktops introduce many more considerations when implementing reliable versions of such algorithms. A naïve algorithm would be appropriate for tiling desktops, but is not appropriate here. Also note that our algorithms nevertheless can be applied in tiling situations, although only expand and ram are applicable.

5. PROCESSING STEP

As we mentioned, we must determine the directions in which the selected window can grow and, for a jostling, ramming, or shoving operation, the directions in which all other windows can move. To accomplish this, we create two related structures: the *desktop graph* and the *grow list*, both of which contain information based on a window classification system.

The desktop graph contains information about *which* windows overlap, *how* they overlap (again, based on the window classification system), and the directions that each window has to move if the if the selected window grows in all four directions. The grow list is then built from this graph, and contains information about the *actual* directions for growth (i.e., the directions that each window has to move based on the specific operation requested). In other words, the desktop graph is a “map” of the desktop, and the grow list is the “directions” for the expand, jostle, ram, or shove. We now discuss, in order, details specific to the window classification system, desktop graph, and grow list.

5.1 Window Classification System

The key to satisfying our guidelines of physically-based and information-preserving growth rests in this classification system. Windows that intersect a growing window *G* must be analyzed for expand and ram (since *G* cannot grow in directions that would cover information in the intersecting window) and jostle and shove (since intersecting windows must be “jostled” or “shoved” in such a way as to maintain their visible contents). Moreover, a group of windows might have to be moved in one direction simultaneously, so their intersection types become critical in maintaining each window’s visible contents. We therefore classify pair-wise overlap (see Figure 6 for a representative of each class).

The numbered labels in Figure 6 indicate what we define as the *number of directions* in which *O* overlaps *U*. Note that cases *zero* and *four* are special cases (i.e. each one is the lone representative for the class) since one window contains the other. The *zero* class is so named since no border of *O* extends past a border *U*; in the *four* class, all four borders of *O* extend past *U*. Since a window that overlaps another window can overlap in at most four directions, the reader may easily verify that the classification is complete.

Each class has a small set of rules for movement (the rules dictate visible-content-preserving *movement* only;

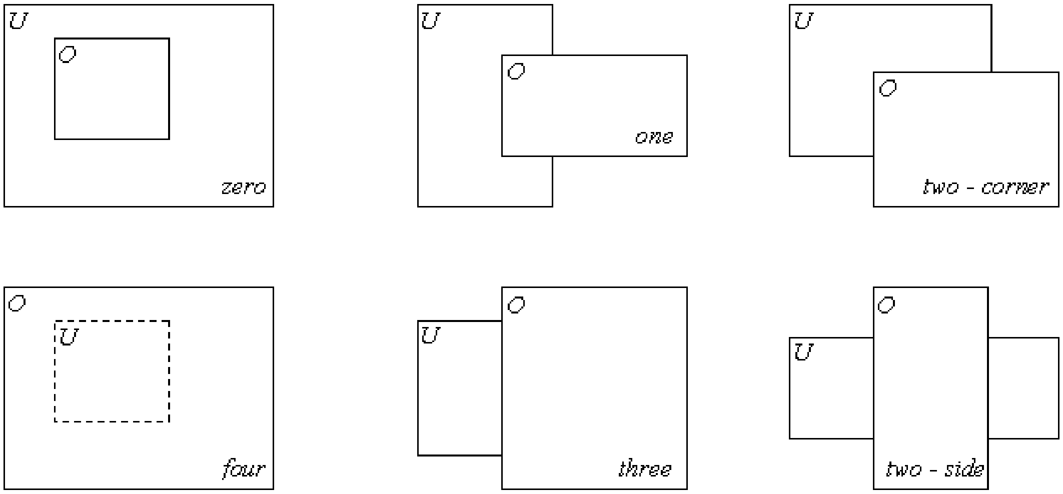


Figure 6: A representative from each of the 6 classes of two-window overlaps. In each example, *O* is the window above (Over) and *U* is below (Under).

growth is discussed later). The general idea is that for a pair of overlapping windows *O* and *U* (where *O* is on top of *U* and thus occludes some portion of *U*), if either window moves in a direction that would occlude visible contents in *U*, then we move the other window in the same direction. Note that ordering of *O* and *U* is important. Consider class *one* and class *three* in Figure 6. If order is ignored, then these classes are equivalent. But in fact, if *O* moves upward in class *one*, then *U* must also move upward (since information in *U* would be covered), yet if *O* moves upward in class *three, U* can remain stationary, since no additional information in *U* would be covered.

We now discuss the rules for each representative of Figure 6. These rules can be generalized for each class, but we use details specific to each representative for ease of explanation. Again, note the use of abbreviations: L for Left, T for Top, R for Right, and B for Bottom.

In *zero*, if *U* moves in *any* direction *d*, then *O* must also move in *d*, as otherwise the visible contents of *U* could change. Similarly, if *O* moves in any direction *d*, then *U* must also move in *d* for the same reason. On the other hand, in *four,* if *O* moves in any direction *d, U* does not have to move, because *U*'s entire contents are invisible at the outset. If *U* moves in any direction *d, O* can also remain stationary, since *U* cannot cover contents of *O*. Note that the processing step notes the set of all windows that are completely covered by a group of windows and subsequently ignores all windows in the set (since the contents are invisible to the user). The method for doing so is straightforward and thus omitted in the interest of space.

For *one*, if *O* moves L, then *U* must move L, as otherwise information in *U* would be covered. Similarly, if *O* moves T or B, then *U* must move T or B, respectively. But, if *O* moves R, *U* remains stationary, since more information from *U* is revealed (similar cases are ignored throughout the rest of the descriptions). If *U* moves T, R, or B, then *O* must move T, R, or B, respectively. For *three* on the other hand, there are only two rules: (*i*) if *O* moves L, *U* must move L; (*ii*) if *U* moves R, then *O* must move R. Note how movement T or B is analogous to "sliding" *O* or *U*.

Sliding also applies to *two-side*: movement of either *O* or *U* in the directions T or B does not affect the other window. Movement of *O* or *U* in the directions L or R, however, implies movement of the other window L or R respectively. Finally, for *two-corner*, if *O* moves L or T, *U* must move L or T respectively; if *U* moves R or B, then *O* must move R or B respectively.

```
for each direction $d$
  initialize $Q$ with the growing window
  while $Q$ is not empty
    $n$ = $Q$.dequeue()
    if any edge $n \to m$ contains $d$ then
      if $m \to m$ does not already contain $d$ then
        add $d$ to $m \to m$
        $Q$.enqueue($m$)
```

Listing 1: Pseudocode algorithm for annotating self-edges in the desktop graph

5.2 Desktop Graph

Having discussed the classification system and associated rules for movement, we now move to the actual processing parts of this step. The first structure that we build is the desktop graph, which essentially encodes the rules for the desktop window configuration in question.

First, we generate a graph node for every window on the desktop. Then we analyze every pair of windows W and X. We draw two directed edges, $W \to X$ and $X \to W$, if and only if W and X intersect. We then annotate every edge with the corresponding rules for movement. For example, if our desktop was representative *two-corner* of Figure 6, then edge $O \to U$ would have annotation {L, T} and edge $U \to O$ would have annotation {R, B}.

So far, the graph tells us how windows must move, but we have not yet determined **if** the windows will actually be moving at all. So we now build the actual moving information for each individual window. To do so, we start by adding a self-edge to every node (i.e., for each window W on the desktop, we add the edge $W \to W$).

The next step requires some abstract thinking. We need to *grow* a selected window G, but our rules are based on *movement*, not growth. But, from the perspective of the other windows on the desktop, a growing window is equivalent to a window that is *simultaneously moving* in all four directions (because, as we mentioned in the previous section, movement is independent in each direction). Therefore, to properly annotate each self-edge, we begin by "moving" G in each direction independently.

The basic idea is to then traverse the graph four times, once each for each direction d. We put G in an initially empty queue (G is the first window to "move"). The queue is used to analyze nodes (windows) that need proper annotations (that will need to move if G "moves"). So, we dequeue a node N, then traverse every edge from N to any other window W. If the $N \to W$ edge annotation contains the direction d (i.e. W must move), then W receives the direction d for the annotation of its self-edge, and W is enqueued. We repeat this process until the queue is empty; the result is that every self-edge $W \to W$ contains the directions that W must move if G grows. Listing 1 gives the pseudocode algorithm for annotation of self-edges.

To see the importance of annotating self-edges, consider the case of Figure 2. For the direction B, we initialize the queue to G. The edge $G \to Y$ contains B, so Y is enqueued. The edge $G \to X$ does not contain B, so X is not enqueued. But we then dequeue Y, and see that the edge $Y \to Z$ contains B, and thus we enqueue Z. Upon dequeueing Z, we find that the edge $Z \to X$ contains B, so X is enqueued. So it turns out that *every* self-edge must be annotated with B (and in fact, every self-edge contains L and T as well). Had we only analyzed the windows that intersect G, then we would have never known that Z would also need to be moved (again illustrating why a naïve approach fails to work properly).

5.3 Grow List

The information that we just built in the desktop graph needs a little more analysis. Reconsider the example given above. The self-edges for X, Y, and Z all contain {L, T, B}. Put simply, "if G grows, then X, Y, and Z must move to the left, top, and bottom." But how does a window move to the top and bottom simultaneously? The answer, of course, is that this is impossible (unless *Z* grows top and bottom; we avoid this since resizing a window can alter its layout and shift previously visible information).

Conflicts in information are resolved by building the grow list. The list contains one element per desktop graph node (i.e. one for each window). The elements contain four values (one for each direction *d*). The possible values are *Must* (the window must move in direction *d*), *Cannot* (the window cannot move in *d*), and *Wait* (the window is not required to move in *d*, but could later move in *d* if requested to do so). Assigning these values is as follows. Initialize each element so that every direction has the value *Wait*. Change each direction that is indicated in the annotation of the element's self-edge in the desktop graph to *Must*. Now, if both L and R have the value *Must*, change both values to *Cannot*, and then do the same for T and B.

6. UPDATE STEP

Now that we have sufficiently described the structure of the desktop through the desktop graph and grow list, we can begin to actually grow and move windows (i.e. to update the positions). We maintain information for movement and growth in two sets: the *available* set and the *restricted* set. The available set determines the directions in which the growing window *G* can grow. The restricted set indicates the directions in which *G* can no longer grow. The need for this distinction becomes apparent later, but it is important now to understand that *G* only grows in the directions that are in the available set.

Since the type of operation (expand, jostle, ram, *or* shove) is simply a parameter to a general algorithm, we discuss each operation separately for each subpart of the update step. The components of this step are (*i*) determining the initial values for the available and restricted direction sets (i.e. the directions for growth) and (*ii*) a cycle of (*a*) updating the available and restricted direction sets and (*b*) actually growing the window *G* and moving the other windows.

6.1 Determining Directions

We use the grow list information to determine the initial directions for growth. The available and restricted direction sets are first initialized to empty. For expand or ram, for each direction *d,* if any element contains a value other than *Wait*, then another window intersects *G* in the direction *d*. Thus, we add *d* to the restricted set. If all the elements have the value *Wait*, we add *d* to the available set.

We can also consider both jostle and shove operations together, since we are only setting the initial directions for growth. The method is similar to that given above, but rather than looking for any element that **is not** set to *Wait*, we look for any element that **is** set to *Cannot*, because the building of the grow list determined that growth in the direction *d* is impossible.

Note how it is thus trivial to initially exclude particular directions, so that if a user wanted to grow in only certain directions, implementation is straightforward (although implementing a good user interface technique is a challenging problem; we discuss a possible direct manipulation interface in the *Future Work* section).

Having built the direction sets, we generate a *moving list* for every direction in the available direction set. The moving list for the direction *d* is populated with every window for which its element in the grow list has the value *Must* for *d*. For expand (and also initially for ram), each of these lists is empty.

6.2 Updating Direction Sets

We now enter the update/grow&move cycle. Before windows are allowed to grow or move however, we have to check for window adjacencies (and of course after windows grow and move we also make this check). So we must verify that the available and restricted direction sets are accurate. For example, in an expand, a direction *d* is moved from the available set to the restricted set if the growing window *G* has (*i*) reached the *d*-edge of the screen or (*ii*) is adjacent to another window in the direction *d*.

```
build the desktop graph and grow list
update direction sets
while (some direction is available)
  if there is a corner touch
    determine the corner
    move and grow in exactly one direction of the corner

  else for each available direction d
    grow the selected window in d by 1
    move all windows in move[d] by 1 in d

  update direction sets
  if intersect relationships changed
    build the desktop graph and grow list
    update direction sets
```

Listing 2: Algorithm for expand, jostle, ram and shove.

A jostle is somewhat more involved. We still employ the same check (*i*) as in expand (since if *G* hits the edge of the screen or a window in the direction *d*, then *d* must be restricted). But since other windows can be moving, additional checks are needed. We consider any direction *d* that is not in the restricted set. If any window *M* in the moving list for *d* has reached the *d*-edge of the screen, we move *d* from the available set to the restricted set. However, if *M* becomes adjacent to a non-moving window in *d*, we remove *d* from the available set, but we do not put *d* in the restricted set. To see why, reexamine Figure 3. *W* begins growing left, thereby shoving *M*. *M* becomes adjacent to *Z*, but then *M* "slides" past *Z*, allowing *W* to resume growth to the left. In cases like this, we once again place *d* in the available set.

For a ram or shove, we again use the same check (*i*) as expand, and then continue to check every direction *d* that is not in the restricted direction set. However, if an adjacency to a non-moving window *N* occurs in *d*, we determine (through desktop graph edge traversal and analysis) which windows must move in *d* when *N* moves in *d*. We also have the potential for an adjacency to be created and then destroyed as in Figure 3 with a jostle. In this case, we remove the windows from the moving list, and then add them back later if another moving window (or the growing window) becomes adjacent.

It is possible that during the course of any of jostle, ram, or shove that two windows that previously overlapped no longer overlap (see Figure 4). In order to preserve the notion of physically-based movement, we must repair the desktop graph and grow list. Therefore, after each availability check, we perform a quick check to verify that all intersections still hold, and rebuild the graph and list if some intersection no longer exists.

6.3 Actual Growth and Movement

The building and verifying of movement sets yields a very straightforward algorithm for actual growth and movement. For each direction *d* in the available direction set, grow the selected window by one unit in *d*. Additionally, move all windows in the moving list for *d* by one unit in *d* (remember that there are no windows in the moving lists for expand).

There is one check that must be made before growth or movement occurs however. There is a possibility that the corners of two windows will touch. In this case, we obviously cannot grow in both directions (as then we would overlap the window, covering part of its contents). Thus, we arbitrarily pick one direction of the corner for movement and move and grow in only that direction. Furthermore, before any movement or growth occurs, we must ensure that the selected window is fully on the screen. This is a simple task, but we mention it to complete the description of the algorithm. We provide listing 2 of the overall algorithm above for further clarity in explanation.

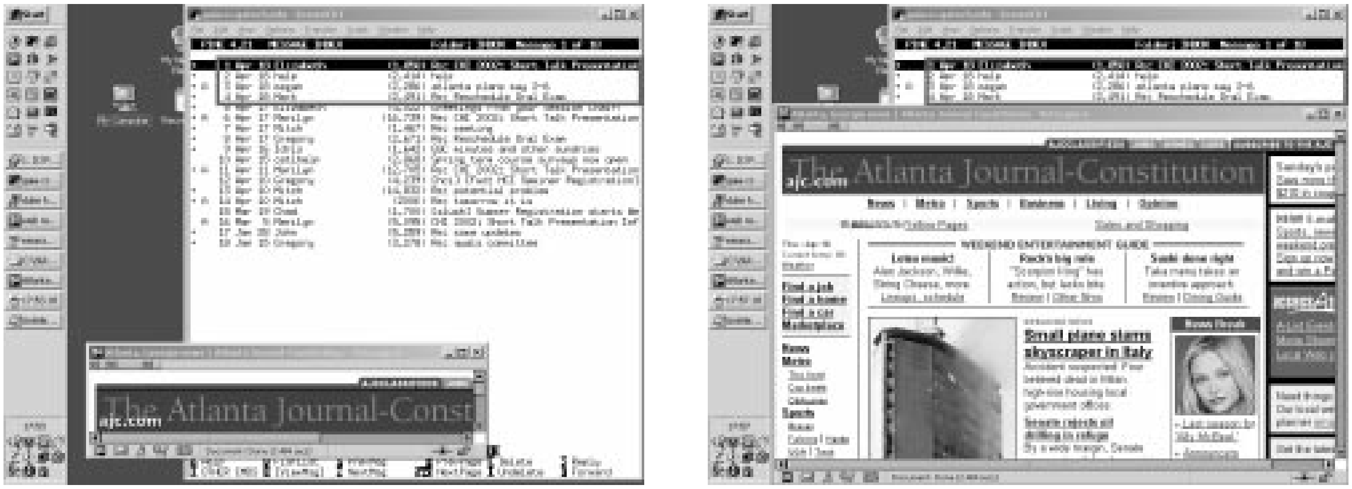
7. CREATING MORE SPACE

For the expand, jostle, ram, and shove operations to be useful, the display space must have some amount of empty space to acquire, as otherwise there is no room in which to grow. Since this may not be the case on any display area, and especially so on a small display, we have developed the concept of a relevant region to attempt to create "virtual" empty space. In each window, the user can define a rectangular region that should always be visible during the operations (for the current implementation, this region is likely to be the part of the window at which a user will occasionally or frequently glance, and is simply indicated my drawing the rectangle with a mouse, but see the *Future Directions* section for possible extensions of the concept). A relevant region then defines the actual border used during the operations. For example, consider Figure 7 below. Displayed is a very simple desktop, with an email client and a newly opened web browser in part (a). The user has outlined a section of the email client in pink (this is the relevant region). This section is where the four most recent email messages appear. Now, when a user issues an expand operation to the web browser, the relevant region acts as the email client's border (see part (b)), thus creating more space for the web browser.

In the current experimental prototype, the default setting is that the entire window is the relevant region, i.e., the actual border should be used for the growing operations. In the future, we would like to explore whether the default should be an empty relevant region (i.e., the window can be ignored during expand, jostle, ram, or shove), or perhaps a region guessed based on a user or task model (although see the *Related Work* section for a discussion of adaptivity). User testing might help to clarify the best choice for the default.

We noted before that we strive to avoid the overhead involved with relationship maintenance, but in a sense, a relevant region defines a relationship on a window. We contend though that this small amount of overhead will not constitute a significant amount of window management time, especially since the relationship is based on exactly one window and not a group of windows, as with constraints. Future use and evaluation should help to clarify the notion and justify this contention.

It may not be obvious how relevant regions fit into the previously described algorithms for expand, jostle, ram, and shove. In fact, though, relevant regions fit quite easily into these algorithms: simply use that defined region as the border of the window, rather than the actual border, *for the bottom window* (whenever two windows are classified).


The Atlanta Journal-Const
ajc.com
The Atlanta Journal-Constitution
ajc.com
Small plane slams skyscraper in Italy

(a) (b)

Figure 7: Both figures are of a simple desktop. In (a) the user has outlined in pink a portion of the email client that displays new massages. This user decides to open a web browser. In (b) the user issues an expand to the web browser, and since the relevant region was defined for the email client, this region is used as the window border, allocating much more space to the web browser than if the region had not been defined.

We must use the region for only the bottom window because, despite the fact that the top window may also have a relevant region, the entire area of the top window is potentially visible and can cover any portion of the bottom window.

Finally, we mention the possibility of automatically shrinking other windows in order to create more space. We do not allow for any shrinking operations in the current implementation. There are several reasons to avoid the shrinking of other windows without an explicit user request to do so. First, windows that are resized are often redrawn by the system, so that small changes in size may give rise to large changes in appearance of the window. Since we strive to maintain the visible portions of windows, we avoid resizing. Another possibility is to simply scale down by a fixed amount. However, it is possible that the smaller representation is unreadable by the user, thus requiring more user maintenance in order to access the information. Finally, we may try an adaptive approach: simply calculate which windows never receive attention by the user, and automatically remove those windows from the screen. As we mentioned in the *Related Work* section, though, a window that is unaccessed by the user is not necessarily unused by the user. Users may use many windows as monitors, frequently glancing at them but infrequently interacting with them (for example a clock, weather display, or instant messenger buddy list).

8. FUTURE DIRECTIONS

There are many directions that we may pursue with this work. The system aspect of the classification and the user-oriented implementation of relevant regions already characterize augmentations to our earlier work [9]. Developing more system-based characteristics (such as more types of classifications or explorations into areas such as empty space representations [4]) may very well lead to new explorations in user-oriented aspects of window management, if not to general theories of windows, as called for by some (for example, see [19]). It is difficult to imagine, however, what "external" aspects may arise from solid "internal" representations without knowing *a priori* the internal representations.

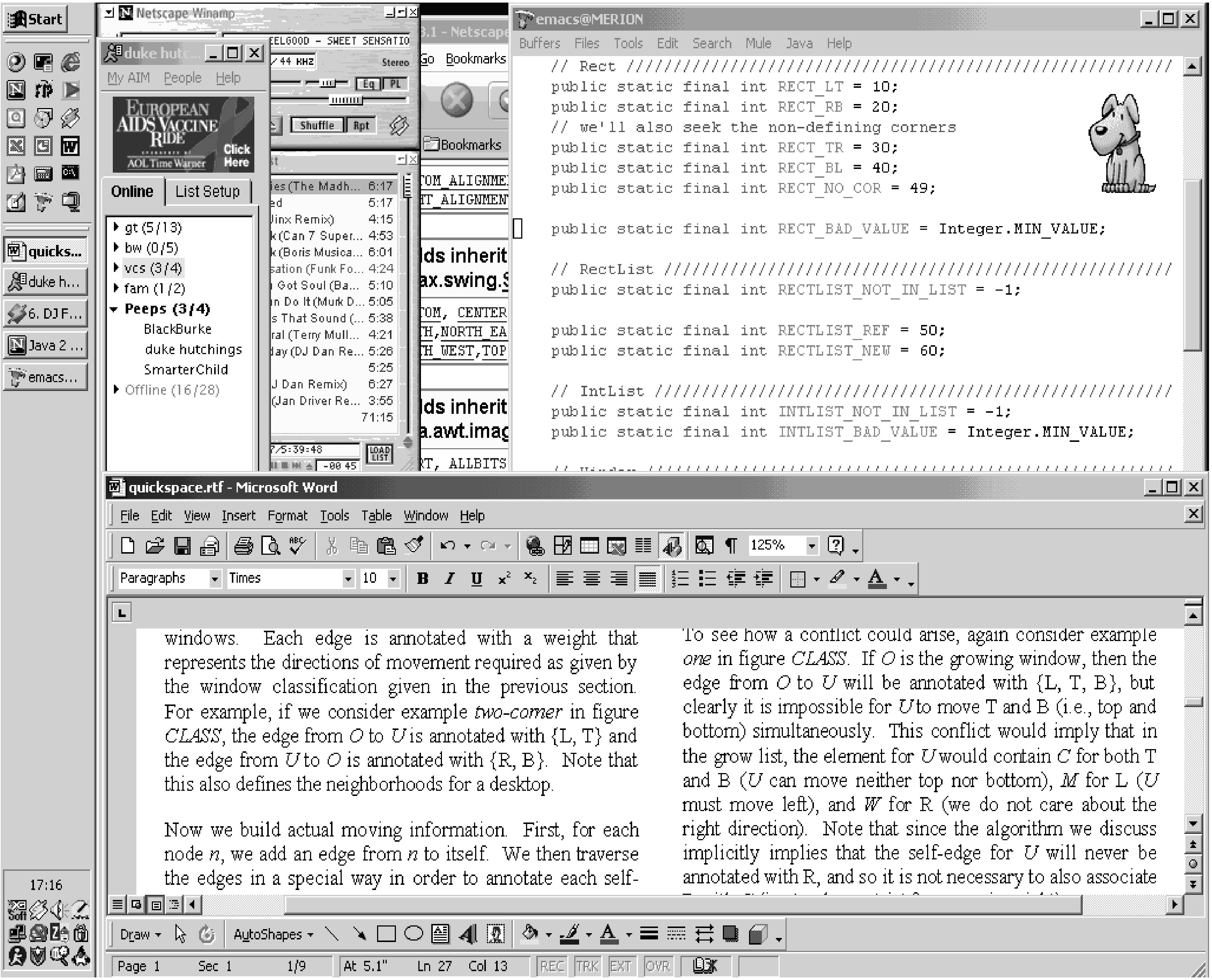
There are a number of potentially valuable enhancements to a user-centered implementation of our current work. One potential disadvantage of expand, jostle, ram, and shove is that the functions assume that a user wants to grow a window in any available direction and that *all* available screen space should be allocated. For example, the user may wish to gain space for a window, but also maintain the window's aspect ratio in order to avoid major reorientation to a new layout of the window's components. This could be accomplished with a slight modification of the algorithms presented. We forgo all of the details here, but the basic idea would be to measure the aspect ratio (m, n) before processing, and then during the update step, rather than growing and moving in each available direction one unit at a time, we move m units in one direction and n units in the orthogonal direction.

The user might also desire to grow in *less* than the amount of space than is available. Consider an "expanding resize" operation, where a user has positioned the mouse in order to resize the window, except in this special case of resize, when the border of another window (or the screen) is reached, further expansion is disallowed. This allows a user to (1) expand shorter than the available distance, (2) quickly determine how far a window can be expanded, or (3) select a particular direction (or corner) in which to expand. As we mentioned, supporting this operation in the current algorithm would be trivial, since all that is needed is the initial restriction of the unused directions.

A "jostling resize," "ramming resize," or "shoving resize" operation, on the other hand, requires more serious thought. Say that a user starts to grow, and then jostle, ram, or shove other windows in an unexpected fashion. It is obvious how to directly revert the growing window to its original position, but not so obvious for the remaining supplanted windows. When two windows collide, there are two possibilities for the resulting relationship between the windows. One is an "oily border" relationship: when the growing window is directed away from the supplanted windows, the windows cease movement. Another possibility is a "sticky border" relationship: when the growing window stops and reverses, so too do the windows that were being supplanted. It is possible to allow this be a user setting, but that raises a question of whether we begin to require too much overhead on the part of the user (sacrificing the guideline of simplicity).

Relevant regions may be further explored as well. Rather than a simple demarcation of an area of a full-sized window, we might actually hide the entire remaining portion of that window and allow the user to interact with only the relevant region as if it were the actual window. This "resize without re-layout" could also help general desktop management, since each window is much smaller and thus possibly more easy to place (compare Figure 8(a) with Figure 8(b)). A successful step in this direction might indicate that more general techniques provided by the window system to allow enhanced interaction with the underlying information and user interface components are in order. This appears to be a relatively unexplored area in window management (since we are concerned with per-window interaction only, not relationships among windows). A *reverse* relevant region may also be useful. The ideas is to hide a section of a window that is irrelevant or annoying (such as an animated advertisement in a web-browser).

Obviously, the directions to pursue can be better understood through evaluation of our existing techniques. We have not yet conducted any formal studies concerning the utility nor usability of our proposed operations. However, we are proceeding carefully in evaluating work in window management. One of the roadblocks to useful and proper evaluation in this field is complete system building: the users for whom these operations would be most useful may be using proprietary window managers (for example, Microsoft Windows). While we could measure such users on other machines, the results may not be as applicable, since these would be operations that people would use for their everyday tasks. But we would not be able to artificially supply such tasks. Moreover, task selection may be a strong factor in determining the results (as indicated, for example, in [10]). In other words, we will have to select a broad cross-section of possible tasks in order to conclude a positive result.



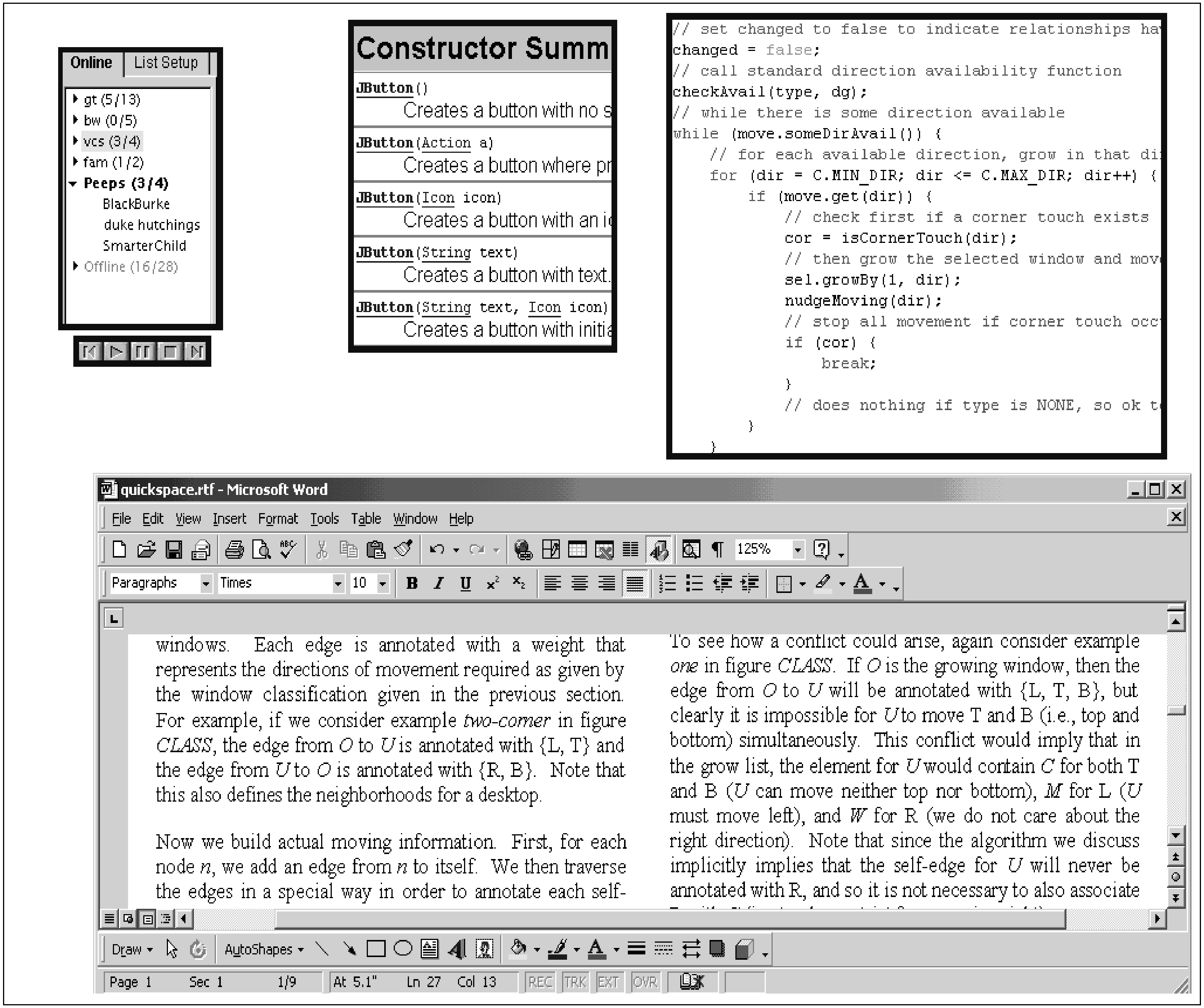
```
// Rect ////////////////////////////////////////////////////////////////
public static final int RECT_LT = 10;
public static final int RECT_RB = 20;
// we'll also seek the non-defining corners
public static final int RECT_TR = 30;
public static final int RECT_BL = 40;
public static final int RECT_NO_COR = 49;

public static final int RECT_BAD_VALUE = Integer.MIN_VALUE;

// RectList ////////////////////////////////////////////////////////////
public static final int RECTLIST_NOT_IN_LIST = -1;

public static final int RECTLIST_REF = 50;
public static final int RECTLIST_NEW = 60;

// IntList /////////////////////////////////////////////////////////////
public static final int INTLIST_NOT_IN_LIST = -1;
public static final int INTLIST_BAD_VALUE = Integer.MIN_VALUE;
```

windows. Each edge is annotated with a weight that represents the directions of movement required as given by the window classification given in the previous section. For example, if we consider example *two-corner* in figure *CLASS*, the edge from *O* to *U* is annotated with {L, T} and the edge from *U* to *O* is annotated with {R, B}. Note that this also defines the neighborhoods for a desktop.

Now we build actual moving information. First, for each node *n*, we add an edge from *n* to itself. We then traverse the edges in a special way in order to annotate each self-

To see how a conflict could arise, again consider example *one* in figure *CLASS*. If *O* is the growing window, then the edge from *O* to *U* will be annotated with {L, T, B}, but clearly it is impossible for *U* to move T and B (i.e., top and bottom) simultaneously. This conflict would imply that in the grow list, the element for *U* would contain *C* for both T and B (*U* can move neither top nor bottom), *M* for L (*U* must move left), and *W* for R (we do not care about the right direction). Note that since the algorithm we discuss implicitly implies that the self-edge for *U* will never be annotated with R, and so it is not necessary to also associate

Figure 8 (a): A somewhat complicated desktop. See figure 8b.



Constructor Summ

JButton()
Creates a button with no s

JButton(Action a)
Creates a button where pr

JButton(Icon icon)
Creates a button with an i

JButton(String text)
Creates a button with text.

JButton(String text, Icon icon)
Creates a button with initi

```
// set changed to false to indicate relationships ha
changed = false;
// call standard direction availability function
checkAvail(type, dg);
// while there is some direction available
while (move.someDirAvail()) {
    // for each available direction, grow in that di
    for (dir = C.MIN_DIR; dir <= C.MAX_DIR; dir++) {
        if (move.get(dir)) {
            // check first if a corner touch exists
            cor = isCornerTouch(dir);
            // then grow the selected window and move
            sel.growBy(1, dir);
            nudgeMoving(dir);
            // stop all movement if corner touch occ
            if (cor) {
                break;
            }
            // does nothing if type is NONE, so ok t
        }
    }
```

windows. Each edge is annotated with a weight that represents the directions of movement required as given by the window classification given in the previous section. For example, if we consider example *two-corner* in figure *CLASS*, the edge from *O* to *U* is annotated with {L, T} and the edge from *U* to *O* is annotated with {R, B}. Note that this also defines the neighborhoods for a desktop.

Now we build actual moving information. First, for each node *n*, we add an edge from *n* to itself. We then traverse the edges in a special way in order to annotate each self-

To see how a conflict could arise, again consider example *one* in figure *CLASS*. If *O* is the growing window, then the edge from *O* to *U* will be annotated with {L, T, B}, but clearly it is impossible for *U* to move T and B (i.e., top and bottom) simultaneously. This conflict would imply that in the grow list, the element for *U* would contain *C* for both T and B (*U* can move neither top nor bottom), *M* for L (*U* must move left), and *W* for R (we do not care about the right direction). Note that since the algorithm we discuss implicitly implies that the self-edge for *U* will never be annotated with R, and so it is not necessary to also associate

Figure 8 (b): A desktop with more open space and is arguably easier to manage (compare with Figure 8a). Rather than showing the whole window for each application, only the relevant parts (selected by the user) are shown (the buddy list of a chat client, the controls for a media player, the constructors for a Java class, and a snippet of an algorithm from a piece of code).

9. CONCLUSION

The intent in composing this paper is to expose the potential vast world of possibilities in display space and window management that have yet to be uncovered by previous approaches, i.e., that window management is not dead yet. By presenting a classification system based on the general properties of windows, and subsequently building some data structures and algorithms, we have presented growth techniques for windows that are simple yet powerful for the user. These physically-based, information preserving, simply executed operations present just a small portion of what could be possible with a more generalized model of window systems and the desktop.

REFERENCES

1. Badros, G.J., Nichols, J., and Borning, A. SCWM: An intelligent constraint-enabled window manager. In *Proc. AAAI Spring Symposium on Smart Graphics* (Cambridge, MA), AAAI press, 76 – 83.

2. Bartram, L., Ho, A., Dill, J., and Henigman, F. The continuous zoom: A constrained fisheye technique for viewing and navigating large information spaces. In *Proc. UIST 1995* (Pittsburgh, PA), ACM Press, 207 – 215.

3. Beaudouin-Lafon, M. Novel interaction techniques for overlapping windows. In *Proc. UIST 2001* (Orlando, FL), ACM press, 153 – 154

4. Bell, B.A. and Feiner, S. Dynamic space management for user interfaces. In *Proc. UIST 2000* (San Diego, CA), ACM Press, 239 – 248.

5. Cohen, E. S., Smith, E. T., and Iverson, L. A. Constraint-based tiled windows. *IEEE Computer Graphics and Applications 6*, 5 (May 1986), 35 – 45.

6. *Freshmeat window manager index.* Available at http://freshmeat.net/browse/56/

7. Funke, D. J., Neal, J. G., and Paul, R. D. An approach to intelligent automated window management. *Int. J. of Man-Machine Studies 38*, (1993), 949 – 983.

8. Henderson, D. A. Jr., and Card, S. K. Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Trans. on Graphics 5,* 3 (1986), 211 – 243.

9. Hutchings, D. R., and Stasko, J. QuickSpace: New operations for the desktop metaphor. In *CHI Extended Abstracts 2002* (Minneapolis, MN), ACM Press, 802 – 803.

10. Kandogan, E. and Shneiderman, B. Elastic windows: Evaluation of multi-window operations. In *Proc. CHI 1997* (Atlanta, GA), ACM Press, 250 – 257.

11. Kandogan, E. and Shneiderman, B. Elastic windows: A hierarchical multi-window world-wide web browser. In *Proc. UIST 1997* (Banff, Alberta, Canada), ACM Press, 169 – 177.

12. Miah, T. and Alty, J. L. Vanishing windows: An empirical study of Adaptive Window Management. In *Proc. Computer Aided Design of User Interfaces* (Louvain-la-Nueve, Belgium), Kluwer Academic Press, 171 – 184.

13. Myers, B.A. State of the art in user interface tools. In *Readings in Human-Computer Interaction: Toward the Year 2000* (Baecker, et. al., eds., 1995), Morgan Kaufman, 323 – 343.

14. Myers, B., Hudson, S. E., and Pausch, R. Past, present, and future of user interface software tools. *ACM Trans. on Computer-Human Interaction 7*, 1 (2000), 3 – 28.

15. Mynatt, E. D., Igarashi, T., Edwards, W. K., LaMarca, A. Flatland: New dimensions in office whiteboards. In *Proc. CHI 1999* (Pittsburgh, PA), ACM Press, 346 – 353.

16. Robertson, G. et. al. Data mountain: Using spatial memory for document management. In *Proc. UIST 1998* (San Fransisco, CA), ACM Press, 153 – 162.

17. Sarkar, M. and Brown, M. H. Graphical fisheye views of graphs. In *Proc. CHI 1992* (Monterey, CA), ACM Press, 83 – 91.

18. Stille, S., and Ernst, R. Adaptive layout calculation to handle the visual chaos in Graphical User Interfaces: A retrospective on the A^2DL-Project. In *Proc. Computer Aided Design of User Interfaces* (Louvain-la-Nueve, Belgium), Kluwer Academic Press.

19. Shneiderman, B. *Designing the user interface: Strategies for effective human-computer interaction.* Addison-Wesley, Reading, Massachusetts, 1998.

20. Storey, M. D., Fracchia, F. D., and Müller, H. A. Customizing a fisheye view algorithm to preserve the mental map. *J. of Visual Languages and Computing 10*, 3, (1999), 245 – 267.