SCWM: An Intelligent Constraint-Enabled Window Manager

Greg J. Badros Jeffrey Nichols Alan Borning
{gjb,jwnichls,borning}@cs.washington.edu
Dept. of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350, USA

Abstract

Typical window management systems rely on direct manipulation techniques to organize and layout windows. Direct manipulation encourages the user to specify particular locations rather than higher-level intentions and desires regarding window layout. Our Scheme Constraints Window Manager (SCWM) allows users to express their intentions using both direct manipulation and higher-level commands. Because some user desires are for persistent relationships to hold among windows, we embed a constraint solver to maintain user-specified constraint-based relationships. To enable using constraints and expressing other layout intentions, we have explored a number of interaction paradigms, including voice recognition. The result is a window manager with much more intelligent window layout and behaviour.

Introduction

Window management systems typically use direct manipulation (Schneiderman 1983) to support window layout. To move a window, the user simply drags the window to the desired location. Similarly, to resize a window, the user grabs a handle at a corner of the window frame and drags it to specify the new desired size.

Sometimes direct manipulation is exactly the right way to manage layout. A user might really want this window to be at a specific position on the display. Often, though, the user has a high-level intention that is being expressed at a lower level by direct manipulation. For example, when a web browser bookmarks window pops up, a user might choose to move it next to the main browser window. The user's intent is not that the bookmarks window be at that specific location on-screen, but rather that the two windows be adjacent. Various layout systems support more accurate placing of windows via a "snapping" mechanism (e.g., snap-dragging (Bier & Stone 1986)). However, those systems still do not capture the true intent that the two windows remain adjacent. Maintaining the relationship requires more sophisticated techniques.

One useful extension to basic snapping is "augmented snapping" (Gleicher 1992). Using this technique, the user has the option of transforming a snapped-to relationship to a persistent constraint that is then maintained during subsequent manipulations. The primary shortcomings of augmented dragging are that the user still needs to place the window initially and that only an adjacency relationship is supported. Without more general techniques, a user cannot have a window remain to the left of another window (but not necessarily attached).

One approach that supports higher-level intentions is to provide the means for users to more directly state their goals. An excellent example is the "maximize window" functionality of many windowing systems. Instead of requiring users to move and resize a window, they can simply click a button to request that the window fill the screen. Window systems that enable better expression of users' desires can be less tedious to use. By allowing users to inform the windowing system about what they are doing at a higher level, the window manager can better support them in the multitude of activities that encompass a typical work session.

We have developed a new X window manager as a platform for research on more advanced and intelligent window layout and interaction paradigms. Our Scheme Constraints Window Manager, or SCWM (Badros & Stachowiak 1999; Badros, Nichols, & Borning 2000b), improves the state of the art in providing smart window layout. We achieve this as a result of two significant design decisions: 1) to embed a Scheme interpreter as its extension language; and 2) to embed the Cassowary constraint solving toolkit (Badros & Borning 1998) to support declarative specification and maintenance of layout constraints. This architecture provides a dynamic, extensible, programmable window manager that enables us to rapidly prototype and experiment with advanced layout techniques and more intelligent interactions.

Higher-Level Window Layout

Ordinary direct manipulation of windows does not preserve enough information for sophisticated window layout. Suppose a user wishes to center a window on the screen. If the window is just dragged to the center of the display, no knowledge of the intention of the user is acquired. This shortcoming becomes a problem when, for example, the user later enlarges the font for that application, thus causing the window to grow. Because a conventional windowing system knows only the absolute position of the top-left corner of the window, it cannot be clever in ensuring that the win-

Copyright © 2000, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

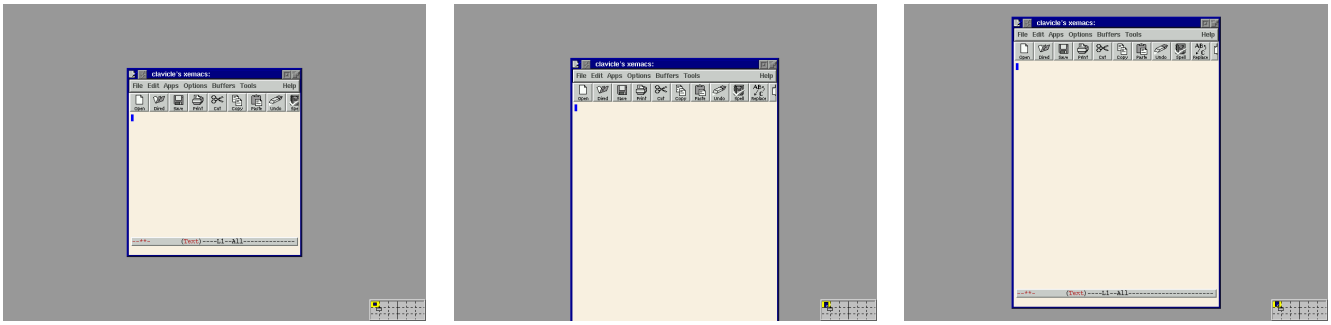

Figure 1: Resizing a window. In the leftmost screenshot, a window appears in the center of the screen. If that window was moved there via ordinary direct-manipulation and the font is then made larger, it might resize by growing to the southeast, as shown in the middle screenshot. However, if the user's intention was that the window stay in the center of the screen, then the rightmost illustration is the desired outcome and is how SCWM behaves after shoving the window to the center.

dow remains centered. With SCWM, the user can instead ask to "shove" a window to the middle of the screen. The windowing system then has the extra knowledge to resize the window appropriately when the user later increases the font size, resulting in the window staying in the center of the screen (figure 1).

Supporting this kind of higher-level information regarding layout is fundamentally important for improving the interaction between users and their windowing systems. Some user desires, such as moving a window to the side of the screen when switching tasks, may require just a single action by the window manager, while others may require persistent behavior, as when the user moves two windows together and expects them to remain connected through subsequent moves and resizes.

For higher-level layout to be of benefit, it must be exceptionally easy and unintrusive for users to express their desires. For advanced users, key-bindings are ideal but difficult to discover without corresponding menu items marked with the key-binding. An attractive alternative is voice recognition. In SCWM, a user can center a window simply by saying aloud "Center current window." The voice recognition interface to window layout and control encourages the user to express higher level intention: it is far more awkward to say "move window to 379, 522" than it is to say "move window next to Emacs." In this way, the voice interface usefully contrasts with direct manipulation where exact coordinates naturally result from the interaction technique. Additionally, voice-based interactions may prove especially valuable for disabled users for whom direct manipulation is difficult.

Since user desires are often for persistent relationships among windows, we provide a user interface that permits declarative specification of these constraints that SCWM then maintains dynamically.

Constraints

Our constraint interface employs an object-oriented design. We specify numerous constraint classes representing kinds of constraint relationships, and zero or more instances of each class are added to the system for maintaining relationships among actual windows. The interface allows users to create constraint objects, to manage constraint instances, and to create new constraint classes from existing classes by demonstration.

Applying Constraints

Applying constraints to windows is done using a toolbar. Each constraint class in the system is represented by a button on the toolbar (figure 2). The user applies a constraint by clicking a button, then selecting the windows to be constrained. Alternatively, the user can first highlight the windows to be constrained, and then click the appropriate button. Icons and tooltips with descriptive text assist the user in understanding what each constraint does. We consulted with a graphic artist on the design of our icons in an effort to make them intuitive and attractive. Preliminary user studies have demonstrated that users can guess the represented relationship reasonably well from the icons even without the supporting tooltip text.

We provide the following constraint classes in our system. Most interesting relationships are either present or can be created by combining classes in the list.

Constant Height/Width Sum Keep the total of the height/width of two windows constant.

Horizontal/Vertical Separation Keep one window always to the left of or above another.

Strict Relative Position Maintain the relative positions of two windows.

Vertical/Horizontal Maximum Size Keep the height/width of a window below a threshold.

Vertical/Horizontal Minimum Size Keep the height/width of a window above a threshold.

Vertical/Horizontal Relative Size Keep the change in heights/widths of two windows constant (i.e., resize them by the same amount, together).

Vertical/Horizontal Alignment Align the edge or center of one window along a vertical/horizontal line with the edge or center of another window.

Anchor Force a window position to stay in place.

Figure 2: Our constraint toolbar. The leftmost button is used to start and stop the constraint-recorder, and the text describes the constraint classes represented by the other buttons in the same order as they are laid out in the toolbar (from left to right).

Some of these constraint types can constrain windows in several different ways. For example, the "Vertical Alignment" constraint can align the left edge of one window with the right edge of another or the right edge of one window with the middle of another. Users specify the parameters of the relationship by using window *nonants* (figure 3). The nonant that the user clicks in dictates the part of the window that the constraint relates. For example, if the user selects the "Vertical Alignment" constraint and chooses the first window by clicking in any of the east nonants and the second window by clicking on its left edge, the resulting constraint will hold the right edge of the first window in line with the left edge of the second. Using nonants makes some constraint classes, such as alignment, more flexible and decreases the number of buttons on the toolbar. Too many narrowly-applicable constraint classes would make the toolbar unwieldy.

NW 0	N 1	NE 3
W 3	C 4	E 5
SW 6	S 7	SE 8

Figure 3: The nine nonants of a window. Clicking in a specific region of a window permits the user to express desires about an edge or corner instead of always meaning the window's center or top-left.

Managing Constraints

Once a constraint is applied, the user still needs to be able to manage it. Users may wish to disable the constraint temporarily or remove it entirely. They may also encounter an odd behavior while they are moving or resizing a window and want to discover which constraint(s) caused the unexpected result. They may simply be curious to know what constraints are applied to a given window and how that window will interact with other windows. Our constraint investigation interface allows for all of these kinds of interactions.

The primary means of inspecting constraints is through visual representations superimposed directly on the windows that the relationship involves (figure 4). When the mouse pointer hovers over a constraint in the investigator, the representation of that constraint is drawn. This feature makes it easy for the user to make the correct associations between windows and constraints. Each constraint class also defines its own visual representation, which in most cases closely matches the icon in the toolbar.

The constraint investigation window also allows the user to enable and disable constraints via a checkbox and to remove constraints via a delete button. The constraint investigator can be kept on-screen at all times and dynamically updates as constraints are applied and removed. Together with the visual representation system, the investigation window makes it easy to manipulate constraints.

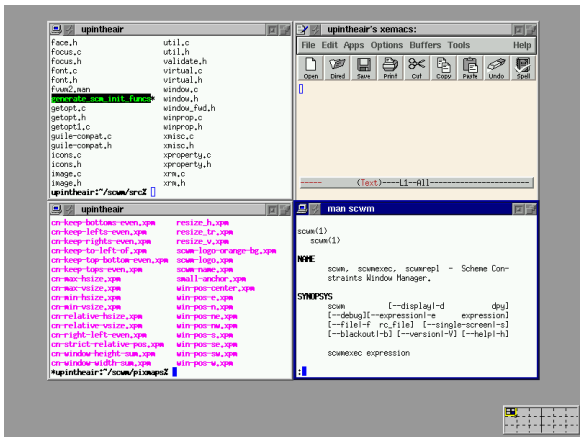

Figure 5: Four windows tiled together. Unlike tiled-only window managers, SCWM permits users to simply tile a subset of their windows; other windows overlap arbitrarily.

A problem with the interface as described thus far is that the basic constraint classes, such as "Vertical Alignment" and "Horizontal Separation," are not always sufficient to convey a user's intention fully. Our own use showed that often one needs to combine several constraints to obtain the desired behavior. A good example of this situation is tiling (figure 5), where two or more windows are aligned next to each other such that they appear to become a window unit of their own. A tiling configuration for two windows can take from three to five constraints to implement. Adding the constraints can become tedious when tiling many windows or when repeatedly tiling and untiling two windows. Certainly a "tiled windows" constraint class could be hard-coded into the system, but that just postpones the problem—some means of abstracting relationships must be provided to the end user.

Our solution to this problem was to create constraint "compositions." A composition is created by a simple programming-by-demonstration technique. We record the

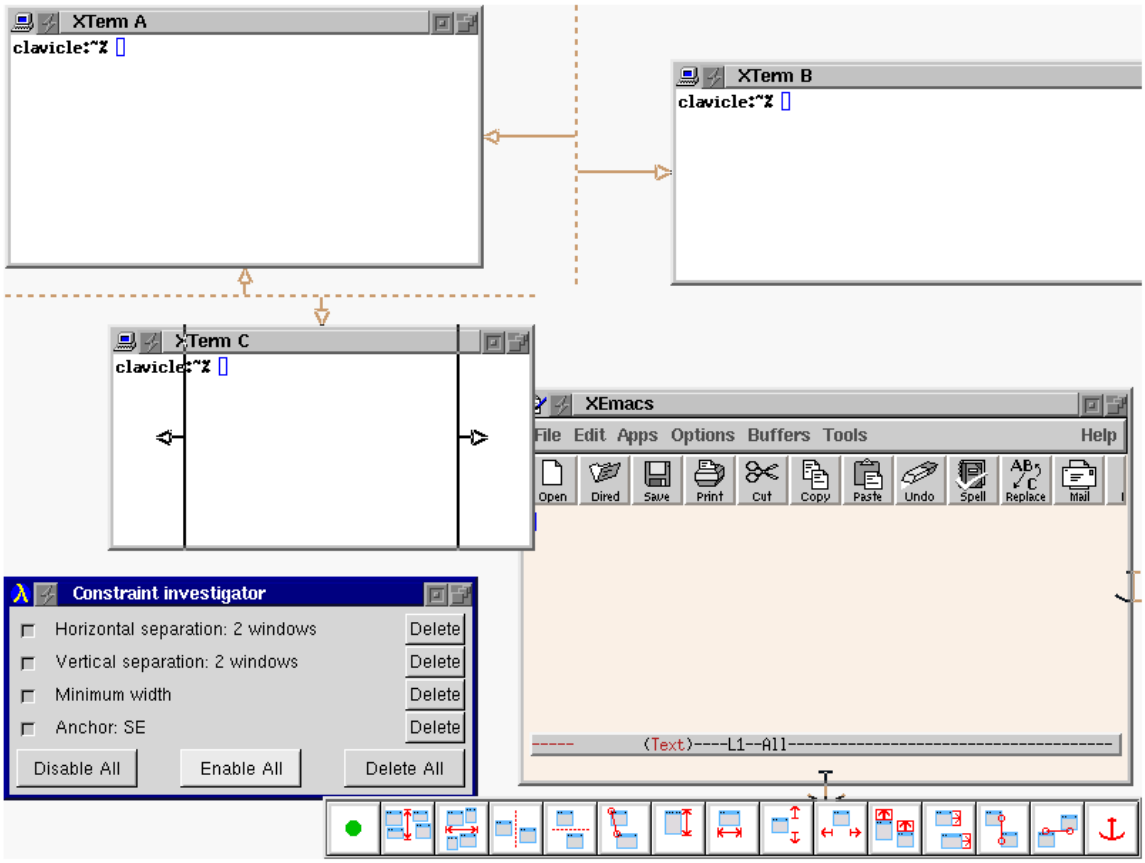

Figure 4: Visual representation of constraints. XTerm A is constrained to be to the left of XTerm B, and above XTerm C. Additionally, XTerm C is required to have a minimum width, and the XEmacs window's southeast corner is anchored at its current location. The constraint investigator that allows users to manage the constraints instances appears in the bottom left of the screen shot.

user applying a constraint arrangement to some windows in their workspace. The constraints used and the relationships created among the windows are saved into a new constraint class object. This class object appears in the toolbar like all other constraint classes. Clicking the button in the toolbar will prompt the user to select the same number of windows as was used when recording. The constraints will then be applied in the same order as before. Compositions allow users to accumulate a collection of often-used constraint configurations that can then be easily applied.

Implicit Constraints

Our graphical interface is not intended to be the only mechanism for adding constraints to the system. Minimizing the effort needed to achieve a desired layout is important. In the ideal case the windowing system would infer useful relationships without requiring interaction from the user. To this end, SCWM supports augmented snap-dragging whereby a strict-relative-position constraint is inferred when windows are placed directly adjacent to one another. SCWM also permits programmers to specify Scheme code that automatically adds constraints when windows first pop up. For example, the procedure that controls placement for windows that are "Netscape: Find" dialog boxes puts the new window at the top right corner of the most-recently focused Netscape browser window and constrains its northeast corner to the browser's northeast corner using the strict-relative-position relationship. Analogous placement procedures can be written for other application types. Ultimately, we expect to extend our interface to permit specifying these kinds of rules at a higher level or to infer them automatically.

Animation for Understanding Constraints

One of the major behavioural changes introduced by using constraint-based window layout is that global rearrangements of windows are possible. In a conventional direct-manipulation interface, the only window that moves is the one currently being dragged. In SCWM, however, moving one window can affect arbitrarily many other windows. Large changes in position are possible, especially when adding and removing constraints. To make these discontinuities less confusing, we animate windows fluidly

from their old positions and sizes to their new configuration.

The animations borrow features from the Self programming environment that mimic cartoon-style animation (Chang & Ungar 1993). As a window is animated to a new position to its left, the window gives a subtle indication that it is active by a quick anticipatory move to the right. It then begins moving to the left, accelerating first, then decelerating to a stop to give the impression of momentum and reinforce the concreteness of the windowing metaphor.

The animations are not limited to constraint effects. When a user shoves a window to the northeast corner of the screen, the window animates smoothly. Similarly, when a window is iconified, an outline of the window shrinks and moves to the position where the icon will rest. Although these features are neither new nor unique to SCWM, they are especially important in helping the user understand the effects of constraints and maintain orientation as global rearrangements occur.

Enabling Technologies

A system of supporting infrastructure enables SCWM to provide all of its sophisticated features while maximizing reuse.

The X Windows System and `fvwm2`

One of the fundamental design decisions for the X windowing system (Nye 1992) was to permit an arbitrary user-level application to manage the various application windows. This open architecture permits great flexibility in the way windows look and behave.

X window managers are complex applications. Many Xlib library functions wrapping the X protocol are specific to the extraordinary needs of window managers. Since the goal of SCWM is to do interesting research beyond that of modern window managers, we used an existing popular window manager, `fvwm2`, as our starting point (fvwm 1999). In 1997 when the first author began the SCWM project with Maciej Stachowiak, `fvwm2` was arguably the most used window manager in the X windows community. It supports reasonably sophisticated configuration capabilities via a per-user `.fvwm2rc` file that is loaded once when `fvwm2` starts. To tweak a parameter, end users edit their `.fvwm2rc` files using an ordinary text editor, save the changes, then restart the window manager to activate the change. The `fvwm2` configuration language supports a very restricted form of functional abstraction, but it lacks loops and conditionals.

Despite these shortcomings, `fvwm2` does provide a good amount of control over the look of windows and has evolved over the years to meet complex specifications (e.g., the Interclient Communication Conventions Manual, or ICCCM (Rosenthal 1994)) and deal with innumerable quirks of applications. By basing SCWM on `fvwm2`, we leveraged those capabilities to ensure that SCWM was at least as well-behaved as `fvwm2`. The fundamental change to `fvwm2` was to replace its ad-hoc configuration language with a version of Scheme called Guile, the GNU project's Ubiquitous Intelligent Language for Extension (FSF 1999).

Scheme as the Extension Language

Guile is a R4RS-compliant Scheme (Clinger & Rees 1991) system designed specifically for use as an embedded interpreter. Scheme is a very simple, elegant dialect of the long-popular Lisp programming language. It is easy to learn and provides exceptionally powerful abstraction capabilities including higher-order functions, lexically-scoped closures, and a hygienic macro system. Guile extends the standard Scheme language with a module system and numerous system libraries wrappers (e.g., POSIX file operations).

Most `fvwm2` commands have reasonably straightforward translations to SCWM symbolic expressions. For example, these `fvwm2` configuration lines:

```
Style "*" ForeColor black
Style "*" BackColor grey76

HilightColor  white navyblue

AddToFunc Raise-and-Stick
+ "I" Raise
+ "I" Stick

Key s WT CSM Function Raise-and-Stick
```

are rewritten for SCWM in Guile/Scheme as:

```
(window-style "*" #:fg "black"
                  #:bg "grey76")

(set-highlight-colors! "navyblue" "white")

(define* (raise-and-stick
          #&optional (win (get-window)))
   (raise-window win)
   (stick-window win))

(bind-key '(window title) "C-S-M-s"
          raise-and-stick)
```

Although Scheme's simple and regular syntax is more convenient for the end user, the greatest advantage of using a real programming language instead of a static configuration language comes from the ability to extend the set of primitive commands and to combine those new primitives arbitrarily.

Adding a new SCWM primitive is easily done by writing a new C function that registers itself with the Guile interpreter. For example, after adding an "`X-property-get`" primitive, we can write:

```
(define*-public
      (window-class
         #&optional (win (get-window)))
  "Return the class of window WIN."
  (X-property-get win "WM_CLASS"))

(bind-key 'all "A-f"
 (lambda ()
   (let* ((win (window-with-focus))
          (class (window-class win)))
     (if (string=? class "Emacs")
         (resize-window 500 700 win)
         (resize-window 400 300 win)))))
```

The above expressions, when evaluated in SCWM's interpreter, will make the user's "`Alt + f`" keystroke resize the window to either 500 by 700 pixels if the currently-focused

window is an `Emacs` application window, or 400 by 300 pixels otherwise.

The preceding example is just a small taste for the power of embedding a Turing-complete extension language in an application. The Emacs text editor was the first application to advocate this architecture and has demonstrated the design's power with its wild success (Stallman 1981). Others have since developed extension languages and further advocated the benefits of scripting for applications (Ousterhout 1998).

The voice recognition module is an excellent example of how our extensible architecture enabled a surprisingly fast implementation of a seemingly-complicated new feature. After getting a sample program from IBM's ViaVoice^{tm} voice recognition engine working, it required less than six hours of development effort to wrap the engine with a Scheme interface and embed it as a dynamically-loadable SCWM module. A grammar describes the various utterances that are understood, and a procedure is asynchronously invoked when a phrase is recognized.

The advantages of SCWM's extensible architecture are even more recognizable in the presence of independently-developed Guile extensions that are then accessible to the window manager. Via standard Guile modules, Scwm can read web pages, download files via ftp, do regular expression matching, and much more. Most significant to SCWM though, is the Guile module that wraps our Constraint Solving Toolkit, Cassowary (Badros & Borning 1998).

The Embedded Constraint Solver

Cassowary is a constraint solving toolkit that supports both arbitrary linear arithmetic equalities and inequalities.[1] Constraints of varying strengths (e.g., required, strong, and weak) can be specified—stronger constraints are satisfied in preference to weaker ones. We implemented the Cassowary toolkit in C++, Java, and Smalltalk and created a wrapper of the C++ implementation for Guile/Scheme. This Scheme wrapper enables us to access the full power of the constraint solver flexibly and dynamically.

To connect the constraint solver with the window manager, the variables known to the solver must relate to aspects of the window layout. Each application window has four constrainable variables: `x`, `y` (the offsets of the window from the top-left corner of the virtual desktop) and `width`, `height` (the dimensions of the window frame in pixels). When Cassowary finds a new solution to the set of constraints, it invokes a hook for each constraint variable whose value it changes; it invokes another hook after all changes have been made. For SCWM, the constraint-variable-changed hook adds the window that embeds that constraint variable to its "dirty set" and the second hook repositions and resizes all of the windows in the dirty set.

To make it easy to express constraints among windows, the constraint variables embedded in each window are available via the accessors `window-clv-{xl,xr,yt,yb,width,height}`, where, for example, `-xl` names the `x` coordinate of the left side of the window and `-yb` abbreviates the `y` coordinate of the bottom of the window.[2] Thus, to keep the tops of two window objects aligned, we can use:

```
(cl-add-constraint solver
   (make-cl-constraint
      (window-clv-yt win1) =
      (window-clv-yt win2)))
```

These primitive constraint-creation constructs are then wrapped by the user interface described in a preceding section to make them accessible to the end user.

Related Work

There is considerable early work on windowing systems (Gosling 1986; Gosling & Rosenthal 1986; Myers 1984; 1986; 1988; Manasse & Nelson 1991). Many of these projects addressed lower-level concerns that a contemporary window manager can ignore. An issue that does remain is tiled vs. overlapping windows. SCWM, like nearly all windowing interfaces of the 1990s, chooses overlapping windows for their generality and flexibility. However, unlike other systems, SCWM's constraint solver can permit arbitrary sets of windows to be maintained in a tiled format of a given size.

Although there are literally dozens of modern window managers in common use on the X windowing platform, only two (besides `fvwm2`) are especially related to SCWM. GWM, the Generic Window Manager, embeds a quirky dialect of Lisp called "WOOL" for Window Object Oriented Language (Nahaboo 1995). It supported programmability, and some of its packages, such as directional focus changing, inspired similar modules in SCWM. Sawmill is a new window manager with an architecture similar to GWM and SCWM (Harper 1999). Like GWM, it embeds its own unique dialect of Lisp (called "rep"). Both embrace the extensibility language architecture and provide low level primitives, then implement other features in their extension language. However, the embedded Lisp dialects used by GWM and Sawmill both suffer from a lack of standardization; worse, GWM's Lisp does not even support the lexical closures that Scheme provides SCWM. Neither GWM nor Sawmill has any constraint capabilities, though the hooks they provide can permit procedural implementations to approximate some of the simpler constraint-based behaviours that SCWM affords.

Elastic Windows (Kandogan & Shneiderman 1996; 1997) is an interesting recent windowing system that uses space-filling tiled layout. The Elastic Windows system does not provide general constraint capabilities, but instead does its layout by the implicit automatically-maintained tiling and a dynamically-alterable hierarchy of windows.

Numerous other application domains have used constraint solvers. Early work includes the drawing tool Sketchpad (Sutherland 1963) and the simulation laboratory ThingLab (Borning 1979). Many other drawing programs

[1] Cassowary also supports a limited set of finite domain constraints, but we currently use that capability only in a constraint-based web browser prototype.

[2] For each window, explicit constraints `xr = x + width` and `yb = y + height` are added automatically by SCWM.

have embedded constraint solvers over the years including Juno (Nelson 1985), Juno-2 (Heydon & Nelson 1994), Unidraw (Helm *et al.* 1995), and Penguin (Chok & Marriott 1998). Unidraw and Penguin both leverage QOCA, a constraint solver that (like Cassowary) is able to maintain arbitrary linear arithmetic constraints (Marriott, Chok, & Finlay 1998).

Web browser layout presents challenges similar to window layout. Our "Constraint Cascading Style Sheets" work also embeds Cassowary and exposes a declarative specification language to web authors for describing page layout (Badros *et al.* 1999). Widget layout in user interfaces is yet another two-dimensional layout problem. Amulet (Myers *et al.* 1997) and the earlier Garnet (Myers *et al.* 1990) both provided constraint solvers based on simple local propagation techniques. These solvers suffer from an inability to handle inequalities and simultaneous equations, which unfortunately arise all too often in the natural declarative specification of layout desires.

Conclusions and Future Work

One of the most useful aspects of this research has been the continuous feedback from our end users throughout the development of SCWM. Since 1997, we have made the latest version of SCWM (along with all of its source code) available on the Internet, and we have actively solicited feedback on our support mailing lists. Many of the high-level layout features were developed in response to real-world frustrations and annoyances experienced either by the authors or by our user community. Although cultivating that community has taken time and effort, we feel that the benefits from user feedback outweigh the costs.

Much needs to be done to continue to improve SCWM and achieve a better understanding of how we can better support the intelligence our users demand from their window manager. We have done some user testing (Badros, Nichols, & Borning 2000a), and more user studies will prove useful to better quantify what benefits users may experience from the advanced features of SCWM.

Our current user interface only supports constraints among windows. It seems useful to permit the addition of "guide-line" elements and allow windows to be constrained relative to them. The user could directly manipulate the guidelines as well, permitting even more flexible layout. These could, for example, be used to ensure that a window stays in the current viewport, or stays in a specific region of the display. Other virtual objects such as "guide-points" may also be useful.

It would also be intriguing to investigate the possibility of ghost-frame objects that are controlled exclusively by SCWM. These window frames could then hold real application windows by dragging them into the frame. This feature would permit hierarchically organizing windows, while still allowing full access to the constraint solver for non-hierarchical relationships.

One of the more interesting complexities of the declarative specification using our current interface is in discerning the user's true intention. This is especially challenging in the presence of windows appearing and disappearing dynamically. Consider a user who is manipulating three windows, **A**, **B**, and **C**. Suppose the user constrains **A** to be to the left of **B**, and **B** to the left of **C**. Now suppose the application displaying in window **B** terminates, thus removing that window. Should window **A** still be constrained to be to the left of window **C**? In other words, should the transitive constraint that was implicit through window **B** be preserved? The answer depends on the user's underlying desire. Providing higher-level abstractions for commonly-desired situations may alleviate this ambiguity. For example, if the user had pressed a button to keep three windows horizontally non-overlapping in a row, it is clear that window **B**'s disappearance should not remove the constraint that window **A** remain to the left of window **C**.

More work also needs to be done to permit the user to interactively describe rules for managing windows of various applications and for applications themselves to automatically add constraints on their various related windows. This sort of application-specific intelligence about how windows should be laid out is best packaged with the individual applications. Thus, it would be useful to provide a framework for X programs to hook into the window manager by injecting Scheme code into SCWM to teach it how to manage its windows.

We also want to extend the use of constraints to encompass additional parts of the window manager's functionality. For example, animations are currently handled procedurally. It would be more consistent and flexible to describe them using constraints relating time and position of window elements (Duisberg 1987). We are planning to integrate a local-propagation based sub-solver into our Cassowary constraint solving toolkit. With access to that subsolver, SCWM users could also use constraints to require windows to iconify together, to require that the title of a window include a string representation of its location, or to require that the color of a window change as it is raised or lowered in the stacking (Z-axis) ordering. An open question is how to better handle non-linear constraints, which are important for some window properties (e.g. area). Differential manipulation may prove useful as one approach to this problem (Gleicher 1994).

Finally, we are especially interested in combining our work with constraints and the web (Badros *et al.* 1999) with this work on window layout. Web, window, and widget layout are all fundamentally related and their similarities should ideally be factored out into a unifying framework so that advances made in any area benefit all kinds of flexible, dynamic two-dimensional layout.

Acknowledgments

We thank Maciej Stachowiak, Sam Steingold, Robert Bihlmeyer, and Todd Larason for their contributions to the SCWM project, and Tom LaStrange, Robert Nation, and Charlie Hines for their work on the window managers that provided the starting point for SCWM. We also thank Kristin Lundquist for her graphic design advice. This research has been funded in part by both a National Science Foundation Graduate Research Fellowship and the University

of Washington Computer Science and Engineering Wilma Bradley fellowship for Greg Badros, and in part by NSF Grant No. IIS-9975990.

References

Badros, G. J., and Borning, A. 1998. The Cassowary linear arithmetic constraint solving algorithm: Interface and implementation. Technical Report UW-CSE-98-06-04, University of Washington, Seattle, Washington. http://www.cs.washington.edu/research/constraints/cassowary/cassowary-tr.pdf.

Badros, G. J., and Stachowiak, M. 1999. Scwm—The Scheme Constraints Window Manager. Web page. http://scwm.mit.edu/scwm.

Badros, G. J.; Borning, A.; Marriott, K.; and Stuckey, P. 1999. Constraint cascading style sheets for the web. In *Proceedings of the 1999 ACM Conference on User Interface Software and Technology*.

Badros, G. J.; Nichols, J.; and Borning, A. 2000a. A constraint interface for managing windows. Short paper submitted for publication. http://www.cs.washington.edu/homes/gjb/papers/scwm-chi-2000.pdf.

Badros, G. J.; Nichols, J.; and Borning, A. 2000b. SCWM—an extensible constraint-enabled window manager. Submitted for publication. http://www.cs.washington.edu/homes/gjb/papers/scwm-usenix2000.pdf.

Bier, E. A., and Stone, M. C. 1986. Snap-dragging. In *Proceedings of SIGGRAPH 1986*.

Borning, A. 1979. *ThingLab—A Constraint-Oriented Simulation Laboratory*. Ph.D. Dissertation, Stanford University. A revised version is published as Xerox Palo Alto Research Center Report SSL-79-3 (July 1979).

Chang, B.-W., and Ungar, D. 1993. Animation: From cartoons to the user interface. In *Proceedings of the 1993 ACM Conference on User Interface Software and Technology*, 45–55. Atlanta, Georgia: User Interface Software and Technology.

Chok, S. S., and Marriott, K. 1998. Automatic construction of intelligent diagram editors. In *Proceedings of UIST 1998*.

Clinger, W., and Rees, J. 1991. *Revised 4 Report on the Algorithmic Language Scheme*.

Duisberg, R. A. 1987. Animation using temporal constraints: An overview of the Animus system. *Human-Computer Interaction* 3:275–307.

FSF. 1999. Guile—The GNU Ubiquitous Intelligent Language for Extension. Web page. http://www.gnu.org/software/guile/guile.html.

fvwm. 1999. The f? virtual window manager. Web page. http://www.fvwm.org.

Gleicher, M. 1992. Integrating constraints and direct manipulation. In *Proceeding 1992 Symposium on Interactive 3D*, 171–174.

Gleicher, M. L. 1994. *A Differential Approach to Graphical Interaction*. Ph.D. Dissertation, Carnegie Mellon University, Pittsburgh, PA. CMU-CS-94-217.

Gosling, J., and Rosenthal, D. 1986. A window manager for bitmapped displays and unix. In *Methodology of Window Management*. Heidelberg, Germany: Springer Verlag. chapter 13, 115–128.

Gosling, J. 1986. SunDew – a distributed and extensible window system. In *Methodology of Window Management*. Heidelberg, Germany: Springer Verlag. chapter 5, 47–57.

Harper, J. 1999. Sawmill. Web page. http://www.dcs.warwick.ac.uk/~john/sw/sawmill.

Helm, R.; Huynh, T.; Marriott, K.; and Vlissides, J. 1995. *An Object-Oriented Architecture for Constraint-Based Graphical Editing*. Springer. chapter 14, 217–238.

Heydon, A., and Nelson, G. 1994. The Juno-2 constraint-based drawing editor. Technical Report 131a, Digital Systems Research Center, Palo Alto, California.

Kandogan, E., and Shneiderman, B. 1996. Elastic windows: Improved spatial layout and rapid multiple window operations. Web page. http://www.cs.umd.edu/users/kandogan/papers/avi96/paper4.html.

Kandogan, E., and Shneiderman, B. 1997. Elastic windows: Evaluation of multi-window operations. *CHI 1997 Proceedings*.

Manasse, M. S., and Nelson, G. 1991. *Trestle Reference Manual*. Digital Systems Research Center. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-068.html.

Marriott, K.; Chok, S. S.; and Finlay, A. 1998. A tableau based constraint solving toolkit for interactive graphical applications. In *International Conference on Principles and Practice of Constraint Programming*.

Myers, B. A.; Giuse, D.; Dannenberg, R. B.; Vander Zanden, B.; Kosbie, D. S.; Marchal, P.; Pervin, E.; Mickish, A.; and Kolojejchick, J. A. 1990. The Garnet toolkit reference manuals: Support for highly-interactive graphical user interfaces in Lisp. Technical Report CMU-CS-90-117, Computer Science Dept, Carnegie Mellon University.

Myers, B. A.; McDaniel, R. G.; Miller, R. C.; Ferrency, A. S.; Faulring, A.; Kyle, B. D.; Mickish, A.; Klimovitski, A.; and Doane, P. 1997. The Amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering* 23(6):347–365.

Myers, B. A. 1984. The user interface for Sapphire. *IEEE Computer Graphics and Applications* 4(12):13–23.

Myers, B. 1986. Issues in window management design and implementation. In *Methodology of Window Management*. Heidelberg, Germany: Springer Verlag. chapter 6, 59–71.

Myers, B. A. 1988. A taxonomy of user interfaces for window managers. *IEEE Computer Graphics and Applications* 8(5):65–84.

Nahaboo, C. 1995. GWM—the generic window manager. Web page. http://www.inria.fr/koala/gwm.

Nelson, G. 1985. Juno, a constraint-based graphics system. In *Proceedings of SIGGRAPH 1985*.

Nye, A. 1992. *Xlib Programming Manual*. Sebastopol, California: O'Reilly & Associates, Inc.

Ousterhout, J. 1998. Scripting: Higher level programming for the 21st century. *IEEE Computer*.

Rosenthal, D. 1994. *Inter-client Communications Convention Manual*, version 2.0 edition. http://www.talisman.org/icccm.

Schneiderman, B. 1983. Direct manipulation: A step beyond programming languages. *IEEE Computer* 16(8):57–69.

Stallman, R. M. 1981. EMACS: The extensible, customizable display editor. Technical Report 519a, Massachusetts Institute of Technology Artificial Intelligence Laboratory. http://www.gnu.org/software/emacs/emacs-paper.html.

Sutherland, I. 1963. *Sketchpad: A Man-Machine Graphical Communication System*. Ph.D. Dissertation, Department of Electrical Engineering, MIT.