USENIX Summer Conference
June 11-15, 1990
Anaheim, California

swm: An X Window Manager Shell

Thomas E. LaStrange – Solbourne Computer Inc.

ABSTRACT

swm is a policy-free, user configurable window manager client for the X Window System. Besides providing basic window manager functionality, **swm** introduces new features not found in existing window managers. First and foremost, **swm** has no default look-and-feel. Like the X Window system itself, **swm** does not dictate policy (look-and-feel); rather, it provides the mechanism for implementing window management policy. Users are not required to learn a new programming language to modify its behavior; instead, simple objects with associated actions determine **swm**'s operation. Its major advantage over other window managers is a feature called the Virtual Desktop. The Virtual Desktop effectively makes the X root window larger than the physical limits of the display and can be panned in a number of ways, including scrollbars, a panner object, or window manager commands. Besides window management, **swm** also provides primitive session management. It can save a user's current window layout and restart those clients when X is restarted. **swm** can restart clients regardless of what toolkit they were built on or what remote host (if any) they were running on. All relevant client information is restored, including window position and size, icon position, and the state of the client.

Introduction

Why another window manager? That is certainly a valid question given the number of window managers currently available for the X Window System. Current window managers fall into two categories: easy to use but not very configurable, or very configurable but complicated to use. Two currently available window managers that illustrate this point are **twm**[LaSt89] and **gwm**[Naha89]. **twm** is easy to use, but different window management policies are next to impossible to implement. **gwm** is policy-free, but requires command of the Lisp language to implement a particular look-and-feel. **swm** shares the best attributes of both of these popular window managers: it is very easy to implement a particular window management policy without the need to learn a new programming language.

swm is also a primitive session manager, a client that can save the current "session" of a user and restart that session sometime later. Users of SunView are familiar with the **toolplaces** program that allows users to move and resize windows and then save the current window layout such that each time they log in, the environment is remembered. This operation is much more difficult in the X Window environment for two reasons: first, client programs may have been built with different toolkits, each having a different set of command line options. Second, clients in the X Window environment are not constrained to be run on the same system that is actually running the X server; in fact, the system on which the client is running may not even be running the same operating system. **swm** solves both of these problems and saves the configuration information in a file suitable to replace the ".xinitrc" file commonly used to store X start-up information.

Architecture

swm is written in C++ using the Object Interface library (OI) [Aitk90]. As will be discussed in following paragraphs, **swm** is object oriented in that it deals with four basic objects to implement window manager appearance and behavior. The small number of objects used by **swm** helps reduce confusion for users. This object oriented approach is facilitated by OI because once a specific object is created, it can be treated as a generic base class object when dealing with attribute settings. This makes the interface to all objects consistent. It not only makes life easier for the user, but internally for **swm** also. When laying out panels, objects can be treated as base class objects without having to know the particular derived type of object actually being manipulated.

Configuration

All of **swm**'s configuration information is specified through the X resource database. This virtually eliminates the need for separate configuration files, such as the .twmrc file required by **twm**, and makes architecture-based configuration much easier. Because **swm** manages multiple screens on a multi-screen X server, there should be some easy way to specify resources on a per screen basis. All **swm** resources begin with the class of the window manager, either "Swm" or "swm", the latter

having precedence. Following that, two strings indicate the screen number and whether or not the screen is monochrome. Some examples might include:

swm.monochrome.screen0
swm.color.screen1

This allows users to define the appearance or behavior of **swm** objects on a per-screen basis if desired.

swm resources fall into two categories: specific and non-specific. Specific resources are those associated with a client window such as an **xclock**. Non-specific resources deal with operational parameters of **swm**. Specific resources differ from non-specific resources in that both components of the WM_CLASS property of the client are included in the resource string. The syntax of specific and non-specific resources is:

Non-specific
swm.*type.screen-number.resource*:
Specific
swm.*type.screen-number.class.instance.resource*:

A full resource specification for the decoration of an **xclock** might look like this:

swm.monochrome.screen0.XClock.xclock.decoration:
noTitlePanel

Using specific resources, one can specify different decorations, icons, behavior, and color for different application classes and instances of classes.

Several template files are supplied with **swm** to get the user up and running quickly. If no **swm** configuration resources have been specified, a default configuration can be loaded. In order to use one of the standard template files, a one line addition to the appropriate X resource file is all that is required. If a user wants to customize **swm**, they can either write their own configuration from scratch, or include and then override defaults in a standard template file. Among the template files are emulations for both the OPEN LOOK and OSF/Motif window managers.

Objects

swm deals with four basic objects: panels, buttons, text objects, and menus. From these four basic objects, an infinite number of window management policies can be implemented. Objects are arranged in hierarchies and can be stacked to any depth. Each object has its own set of attributes, such as color, font, and cursor. Each object also has a "bindings" attribute that describes the actions to be taken when mouse buttons or keyboard keys are used while the pointer is in the object.

Panel Object

The panel is the most basic object in **swm**. It is nothing more than a container for other objects. Objects within panels are organized into rows. A row within a panel is made up of one or more other objects. Panels can be divided into five distinct classes:

1. Decoration
2. Icon appearance
3. Root icons
4. Root panels
5. Icon holders

The syntax for defining a panel is the same regardless of what panel class is being created. The syntax specifies the objects within the panel and how they are positioned. The syntax for a panel definition is:

swm*panel.*panel-name*: \
object-type *object-name* *position* \
object-type *object-name* *position* \
object-type *object-name* *position* \
.
.

panel-name is the name used to reference this panel. Each object within a panel has three components: *object-type* defines the type of object created. Valid object types include **panel**, **button**, and **text**. *object-name* defines the name used to reference the subcomponent; and *position* is a geometry string describing the position of the object within the panel. The X and Y components of the geometry map to the column and row position within the panel.

Decoration Panels

Decoration panels describe what client windows look like after they are reparented; they describe the "decoration" the window manager places around given client windows. The syntax for specifying a decoration panel is exactly like any other panel

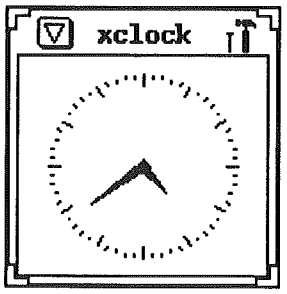

Figure 1: OpenLook+ Decoration

except that it contains an interior panel object called "client." For example, the definition below specifies the default decoration for a client window in the OpenLook+ template, which is supplied with **swm**. Figure 1 shows what a decorated window looks like using this panel definition.

```
Swm*panel.openLook: \
  button  pullDown  +0+0 \
  button  name      +C+0 \
  button  nail      -0+0 \
  panel   client    +0+1
Swm*panel.openLook.resizeCorners: True
```

The openLook panel defined above is made up of four objects. In the above example, the button object called "name" has the letter "C" as its column coordinate, meaning to center the object within the row.

As noted earlier, the decoration panel must contain a panel object called "client." This special panel is where the client window is placed in the decoration panel. In the above example, the client window is placed in column 0 of row 1. Keep in mind that because the "client" panel is a generic panel object, there are no limits on what a decoration panel must look like. Objects can easily be placed to the sides or below the client window in addition to the more traditional "title-bar" appearance.

Another special object that can be used in a decoration panel is a button or text object called "name." This object displays the WM_NAME property of the client and can be seen in the above example. The "pullDown" and "nail" objects provide other functionality for this panel.

Icon Appearance Panels

swm has no concept of what an icon should look like; it is up to the user to describe how icons should be represented. Icon appearance panels serve this purpose. The default icon of the OpenLook+ template is defined as follows:

```
Swm*panel.Xicon: \
  button  iconImage  +C+0 \
  button  iconName   +C+1
Swm*defaultIconImage: @xlogo32
```

In the above example, the Xicon panel contains two buttons, both of which have special meanings. The iconName button displays the contents of the WM_ICON_NAME property. If the client has specified a pixmap to display as the icon or has specified its own icon window, that image is displayed in the iconImage button. In our example, if the client has not specified an icon, the iconImage button will contain the image of the xlogo32 bitmap file.

As with the decoration panels, there are no limits on how complex icon appearance panels may be.

Root Icons

The Icon Appearance panels described above are used when a corresponding client application has been placed in an iconic state. Root icons are simply icon appearance panels that do not correspond to any client application. Because of this they cannot be deiconified. Since they are icons, they can be moved, have bindings associated with them and can use icon holder panels (see below). What good are they? In particular, they can have bindings describing actions such as what should happen when they are the destination of an operation such as drag-and-drop.

Root Panels

Root panels are static panels (usually containing buttons) that are always visible on the root window. One can think of a root panel as a menu that is always visible. Root panels differ from Root icons in that they are treated like other client windows, i.e., they get reparented, can be iconified, etc. Figure 2 shows a reparented root panel and its definition:

```
Swm*panel.RootPanel: \
  button  quit      +0+0 \
  button  restart   +1+0 \
  button  iconify   +2+0 \
  button  deiconify +3+0 \
  button  move      +0+1 \
  button  resize    +1+1 \
  button  raise     +2+1 \
  button  lower     +3+1
```

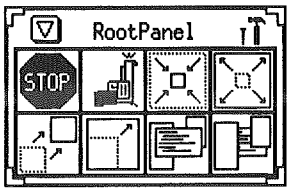

Figure 2: Root Panel Example

Icon Holder Panels

Icon holder panels are special root panels that contain icons. They provide an optional scrolling window in which icons can be deposited and managed. Icon holder panels are similar to the Icon Manager feature of **twm** but more flexible in that actual icons are managed rather than a fixed-appearance icon representation. Icon holder panels have many options, including being not displayed when empty, and sizing to fit all the icons rather than presenting a scrolling window. Icon holder panels can be created to contain specific classes of client icons. In this way you can do things such as group all **xterm** icons in one panel, and other icons in a separate panel.

Button Object

The button object can contain either text or a bitmap image. Buttons are heavily used in defining window manager decoration and icon appearance. The button object is unique in that its appearance can be changed dynamically through the use of window manager functions. This allows window manager decorations to change depending on the state of the

client application. Like other **swm** objects, the button object can also have its bindings (functions) changed dynamically. With these two features, buttons can not only dynamically change appearance, but they can also change functionality.

Object Attributes

Each object, when created, queries the X resource database for a number of attributes. These attributes include color, font, cursor, and bindings. Among these attributes, the bindings attribute is the most interesting. The binding attribute defines the behavior of the window manager. Most other window managers, provide mouse and keyboard bindings in a particular context, i.e., window manager frame, icon, or client window. In **swm** you can think of each object as being its own context with its own set of actions. **swm** does not know that an object is within a window decoration panel or an icon, all it knows is that some object requested that a specific action take place when a mouse button or key event is detected within it. Object bindings are specified in an X Toolkit Intrinsics[McCo]90 format so that those familiar with the Xt syntax will not have to learn yet another way of specifying actions:

```
swm*button.foo.bindings: \
    <Btn1>     :  f.raise \
    <Btn2>     :  f.save f.zoom \
    <Key>Up    :  f.warpvertical(-50)
```

This example shows bindings for a button object called "foo." Assuming that the button is in a window manager decoration panel, if mouse button one is clicked, the window is raised to the top of the stack. If mouse button two is clicked, the window's location and size are saved and the window is expanded to the full size of the screen. If the Up key is pressed while the pointer is over the button, the pointer will be warped "up" 50 pixels. It is useful to note that any number of bindings can be specified and that any number of functions can be specified per binding.

Window Manager Functions

All window managers provide functions allowing the user to do basic operations such as moving and resizing windows. In **swm**, functions are specified via a bindings attribute as shown in the previous paragraph. **swm** functions can be invoked in several modes. Using the **f.iconify** function as an example, here are the possible ways to execute the command:

f.iconify
  Iconify the current window
f.iconify(multiple)
  Iconify multiple windows, prompting for each window
f.iconify(blob)
  Iconify all windows whose class matches "blob"
f.iconify(#$)
  Iconify the window under the mouse
f.iconify(#0x1234)
  Iconify a particular window ID

Another important feature of **swm**'s command execution is that commands can be executed on behalf of an outside client. By writing a special property on the root window, **swm** interprets its contents and executes commands. A simple client has been written called **swmcmd** that provides a way to execute window manager commands by typing them into a shell running in an **xterm**. This provides a mechanism for executing any command at any time; it is particularly useful when you have forgotten to add the command to one of your pop-up menus. By simply moving your pointer into any **xterm** and typing:

```
swmcmd f.raise
```

The pointer would be changed to a question mark prompting you to select a window to be raised. This interface could also be used for things such as changing the shape of a button to indicate the status of a process.

SHAPE Support

swm provides support for the SHAPE extension, allowing non-rectangular windows. Each object can have a separate shape mask attribute which is simply a bitmap image of the shape of the object. In addition to the shape mask, if a panel object is to be shaped and no shape mask is specified, it is shaped to contain its children.

swm recognizes if a client window is shaped and adds the string "shaped" to the beginning of the resource strings used to obtain attributes. This allows attributes to be specified for shaped client windows. A useful example would be to provide shaped decoration for shaped clients:

```
swm*shaped*decoration:  shapeit
swm*panel.shapeit:  panel  client  +0+0
swm*panel.shapeit*shape:  True
```

This particular decoration allows invoking the X11R4 **oclock** or **xeyes** clients and they would be displayed without visible decoration.

The Virtual Desktop

Probably the single most exciting feature provided by **swm** is the "Virtual Desktop." This feature effectively makes the X root window larger than the physical limits of the display. This large "root" window can be panned using scrollbars, a two dimensional panner object, or window manager functions. Using the Virtual Desktop, it is very easy to implement a "rooms" like environment by grouping windows into various quadrants of the desktop.

Virtual Desktop Panner

The panner object provides several important features of the Virtual Desktop.

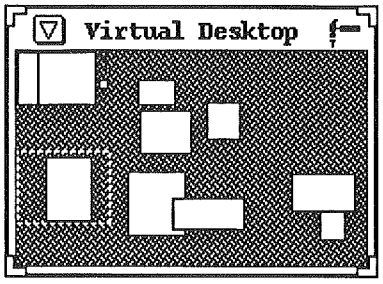

Figure 3: Virtual Desktop Panner

The panner shows a miniature representation of all windows currently on the Virtual Desktop. It also displays an outline indicating your current position within the desktop. When the pointer is in the panner and mouse button one is pressed, the current position outline can be moved to view another portion of the desktop. If mouse button two is pressed while the pointer is over one of the miniature windows, a move operation is started on the window. A small outline appears under the pointer representing the window shape that is being moved; it can be dropped anywhere in the panner and the actual client window is repositioned. If the pointer is moved out of the panner during the move operation, a full size outline of the window is displayed, allowing the user to move and fine tune the placement on the current visible portion of the desktop. This feature also works when the window move was started on a client window and the pointer is moved into the panner, allowing the user to move the client window to any portion of the desktop.

The panner is reparented so it can be moved, iconified, and resized just like any other client window. The act of resizing the panner object causes the underlying Virtual Desktop window to resize. This allows resizing the desktop at run time depending on the user's needs. Because the Virtual Desktop is an X window different from the actual root window, the size of the Virtual Desktop is limited only by the usable area of an X window, 32767 x 32767 pixels.

Sticky Windows

One might wonder why it is that the panner does not scroll off the display when the desktop is panned. In conjunction with the Virtual Desktop, **swm** introduces a feature called "sticky" windows. Sticky windows appear as though they are stuck to the glass of the display. When the Virtual Desktop is scrolled, sticky windows do not move. They allow the user to set up a standard environment that can be used regardless of the current position of the desktop. This standard environment might contain things such as a clock and mail notifier, which would then be visible no matter which portion of the Virtual Desktop is being viewed. Windows can be stuck and unstuck interactively by the user. Classes of windows can also be specified to start up as sticky and decorations can be dependent on whether or not the client window is sticky. Consider the following resources:

```
swm*xclock*sticky:    True
swm*sticky*decoration:    stickyPanel
```

These resources tell **swm** that any client that has the class "xclock" should start up as a sticky window. When **swm** finds that it is reparenting a sticky window, it places the string "sticky" in the resource string so that decoration and other attributes can be dependent on whether or not a client window is sticky.

Virtual Desktop vs. ICCCM

The Inter-Client Communications Conventions Manual (ICCCM)[Rose89] outlines a specific client/window manager communication protocol. The Virtual Desktop provides many challenges to both **swm** and clients in deciding exactly what should be done to remain ICCCM compliant.

Window Positions

Window positioning is probably the most common problem applications will encounter when running under **swm** with the Virtual Desktop enabled. Many clients monitor their position on the root window to allow intelligent positioning of dialog boxes or popup menus. When clients are moved to other portions of the desktop, they get positioning information which leads them to believe that they have been moved off the visible portion of the root window. When popping up a dialog box, the client may detect that its window is off the screen and attempt to position the dialog box back on the screen. This may cause it to be displayed in the upper-left quadrant of the desktop, possibly making it nonvisible.

Besides large position coordinates, other problems arise when the desktop is panned. If the desktop is positioned at 0,0, a window at location 100, 100 on the desktop is also at location 100, 100 relative to the real root window. If the desktop is panned to location 25, 25, the client window is still at location 100, 100, with respect to the "root" window onto which it has been placed, but is now at location 75, 75 with respect to the real root window. The window gets no ConfigureNotify events, real or synthetic, because it hasn't moved with respect to its root. If the client window is monitoring its position for use in popping up a menu or dialog box, its position with respect to the real root has changed and the popup window will probably be placed improperly.

swm and the OI toolkit have attempted to solve these problems. When **swm** reparents a window it places a property on the window indicating the

window ID of its "root" window. This will be the window ID of the real root window or the ID of the Virtual Desktop window. The toolkit then reparents, maps, and positions popup menus and dialog boxes with respect to the window ID specified in the property rather than always using the actual root window. In addition, this property is updated whenever the root window for a client changes. This occurs, for example, when a window is made sticky or unsticky. Besides solving the window positioning problems, this would also allow **swm** to implement multiple Virtual Desktops, although at this point in time we're not sure how useful such a feature would be.

USPosition vs. PPosition

The ICCCM specifies two methods for X clients to request window positions: User Specified Position (USPosition) or Program Specified Position (PPosition). Both of these flags exist in the WM_NORMAL_HINTS property. These hints allow the user or application writer to request window positions that may or may not be honored by the window manager. The idea behind these hints is that if a user requested a window at a given location, it would presumably have priority over a default window location requested by the X client, which in turn might have priority over default window manager placement. Previous to X11R4, the X Toolkit Intrinsics always set the PPosition hint, effectively making the hint useless as all window managers simply ignored it.

swm uses these hints to determine window placement on the Virtual Desktop. If USPosition hints are specified, the window is placed at the absolute location requested by the user, even if the coordinates on the desktop are not currently visible. If PPosition hints are specified, the window coordinates are assumed to be relative to the current visible portion of the Virtual Desktop. For example, if the desktop is positioned such that the upper left corner of the display corresponds to pixel location 1000, 1000 on the desktop, a USPosition of +100+100 would place the window at 100, 100 on the desktop. If a PPosition of +100+100 is used, the window would be placed at 1100, 1100.

Because of the problems mentioned previously, multi-window applications wanting a default window layout usually set the USPosition hints to achieve their layout. While in most cases this presents no problems, it has the effect of making the application usable only in the upper-left quadrant of the Virtual Desktop, something many users will no doubt find objectionable.

Application writers needing a default window layout should make use of the PPosition hints and only use USPosition if the window positions have been requested by the user, either through the X resource database or through command line options. If compatibility with older window managers is desired, an option should be added to such applications so that the older behavior can be selected through a "bug compatibility" mode or something similar.

Session Management

There does not currently exist a standard mechanism in X for a user to save a "session" and then later restart the session. The most desired type of session management is one in which the locations of all client windows are saved, plus the state of each client. This would allow a window based editor to be shut down and brought up at some later date, editing the same files at the same positions within the files. Within the next few years this type of session management will gradually be designed, proposed, discussed, revised, and finally implemented. Until this happens, users have absolutely no standard way to do even basic session management, such as that provided by the SunView **toolplaces** program.

There are several reasons why even simple session management in X is difficult. First, since there is no standard toolkit, there are no standard command line options. The **xplaces** client attempts to do simple session management but assumes that X Toolkit Intrinsics options are used. This leaves users of the XView toolkit or other non-intrinsic based toolkits out in the cold. Second, client programs are not constrained to run on the same host as the X server and may even be using different operating systems.

swm attempts to solve both of these problems. It implements session management using a two step approach: first, an **swmhints** program provides **swm** with hints about the client's previous state; and second, **swm** interprets those hints to restore client windows to their previous state and location. The **swm** command **f.places** causes a file to be written which can be used as an .xinitrc replacement. This file contains two lines for each client to be restarted: an **swmhints** invocation with appropriate options, and the actual invocation of the client. The client is invoked with the exact command string found in the WM_COMMAND property. This allows restarting of clients regardless of the command line options they understand. An example client startup script might look like this:

```
swmhints -geometry 120x120+1010+359 \
  -iconGeometry +0+0 -state NormalState \
  -cmd "oclock -geom 100x100 "
oclock -geom 100x100 &
```

This example shows that the client **oclock** was originally started with a geometry of 100x100. Sometime later it was resized to 120x120 and positioned at location 1010, 359. If iconified, its icon would be positioned at 0, 0. Its state is NormalState as opposed to iconic. The **-cmd** option to **swmhints**

simply echoes the WM_COMMAND property string looked for by **swm**. All of the information given to the **swmhints** program is appended to a property on the root window. When **swm** starts up, it reads this property and stores the information in an internal table. When a client window is reparented, the table is searched for a matching WM_COMMAND string and possibly a matching WM_CLIENT_MACHINE property. If a match is found, the entry is removed from the table and the window is sized and positioned to its old location. When restarting a client, **swm** restores the following attributes to the client window: window size, window location, icon location, whether or not the icon was on the root window, window sticky state, and the normal or iconic state of the window.

The scheme outlined above breaks down if two windows have identical WM_COMMAND properties. In this case, **swm** cannot distinguish between the two windows as they are being reparented. In practice this does not appear to be a problem for most users.

Restarting Remote Clients

Remote clients provide a challenge for session management. In a UNIX environment, if a user performs an **rlogin** to a system and then starts a client by simply typing the client name with no options, the client starts correctly if the DISPLAY, PATH, and any other environment variables needed by the client are set properly. In this case the WM_COMMAND property contains the client name and the WM_CLIENT_MACHINE property contains the name of the machine on which the client is running. This information by itself is not sufficient to restart the client. If the shell being used only reads an initialization file for login shells, it is very possible that the PATH and DISPLAY environment variables of the shell used for the remote startup will not be set or will not contain enough information for the client to run properly. **swm** provides the user with a resource that allows a customizable string to be used when starting remote clients. This string could be set up to include PATH and DISPLAY environment variable settings although this may limit portability when using the same startup file on different machines.

Evaluation

swm was developed, tested, and refined in approximately nine man-months. This relatively short development time was due to two factors: previous window manager design experience, and use of the OI Library. While this library provides the objects and interface to the window system that make it easy to write applications, there is certainly some performance penalty to be paid because of the extra overhead. **swm**, like any toolkit based window manager, has somewhat slower performance than a window manager written directly on top of Xlib or one that is kernel based. However, the added functionality and flexibility is well worth the speed trade-off, especially with the performance of newer machines.

One of the biggest mistakes made with **twm** was using a separate initialization file rather than the more general X resource database for configuration. The X resource database provides the user with many more configuration options, and makes extending **swm**'s capabilities easier.

Conclusion

In the introduction I posed the question, "Why another window manager?" The answer is relatively simple. The battle between user interfaces has not yet been won. **swm** was developed with this in mind. It is able to adapt to most any user interface. It also introduces new features, such as the Virtual Desktop, that are designed to improve user productivity. In addition, these features can all be configured by a standard method, the X resource data base.

References

[Last89] LaStrange, Tom, "An Overview of twm (Tom's Window Manager)," Xhibition '89 Conference Proceedings.

[Naha89] Nahaboo, Colas, "GWM, The Generic X11 Window Manager," Xhibition '89 Conference Proceedings.

[Aitk90] Aitken, Gary, "OI: A Model Extensible C++ Toolkit for X Windows," 1990 X Technical Conference proceedings.

[McCo90] McCormack, Joel, Asente, Paul, and Swick, Ralph, "Xt Toolkit Intrinsics - C Language Interface," X11R4 documentation

[Rose89] Rosenthal, David, "Inter-Client Communication Convention Manual," X Window System documentation

Tom LaStrange received a B.S. in Computer Science from the University of Utah in 1981. He got involved with X while working at Evans & Sutherland where he developed graphics microcode for the DEC VaxStation 8000 workstation to support X11R1. During his time at Evans & Sutherland, he also developed the initial version of the **twm** window manager for X. This window manager is now supported by MIT as the standard X window manager for X11R4. Following Evans & Sutherland, he worked for Hewlett Packard's Graphics Technology Division where he was involved in optimizing X server performance for HP's high-end

graphics workstations. He is currently working in the User Interfaces and Applications group at Solbourne Computer, where he is involved in OI toolkit development and application design, as well as his work on **swm**. Reach Tom at 1900 Pike Road, Longmont, CO 80501. His electronic address is toml@Solbourne.COM.

USENIX Association

Proceedings of the Summer 1990 USENIX Conference

June 11 – June 15, 1990
Anaheim, California, USA