copyright as X11 since July 1989. It is in MIT X11R4 Release, and we plan to further enhance it. The WOOL language itself is used in other projects (an Interface Builder and graphical objects system) and might be released as a stand alone product.
Most GWM users feels that it is as as fast as other non-extensible window managers. It is coded in standard C, and has been ported to numerous machines (DEC Vax, sun 3 & 4, BULL DPX1000, Apollos, Gould powernodes, IBM PC/RT, Sony News...)
Our hope that GWM will encounter the same success as EMACS, might still become a reality...

When an event is sent to a *fsm*, if it is an "housekeeping" event such as `Expose` or `MapNotify` events, it is directly handled by GWM, otherwise it is matched against all the transitions in the current state of the fsm, and as soon as one matches, the corresponding WOOL expression is evaluated, and the fsm changes state if necessary. If no transition is waiting for the event, it is discarded.

4 Validity of our design choices

One year and a half after the first version of GWM (March 1988), we can look back at our design choices and ponder whether we were right.

no policy does not hamper people in designing complete packages, but prevents the end user to just add a friend decoration to his profile if they are designed by two different programmers. GWM needs some programming rules, not unlike the Toolkit Intrinsics, to allow easy interchanges of our "widgets", the window decorations, icons, menus, and desktop space management algorithms. This is the area were most progresses must be made, hopefully with the help of the growing GWM community, to enhance the *standard profile*.

wob descriptors were a kind of cache to allow pre-construction of wobs. We were trying to reduce the execution of WOOL code, because we expected it to be slow. In fact WOOL fast enough to get rid of this added complexity, and they will disappear in the next major version of GWM. Wob constructors will directly create the objects and associated windows.

bars around a window approach simplified the geometry problems, but now clearly limits what can be done with GWM[10]. We must now provide fully recursive compound objects (now bars cannot contain bars, only plugs!).

not toolkit based is really a good choice. Window Managers needs capabilities not needed by normal applications, such as multi-screen management, that toolkits do not implement yet.

custom lisp dialect is something we don't regret, seeing the good performances of WOOL, although it is somewhat hard for GWM to learn a new language with its own idioms.

custom object oriented C layer was used instead of using a truly object-oriented language. Though not very sure at first of the validity of this approach (as opposed to using C++), the extreme portability of GWM fully justifies it now.

Globally, most GWM flaws stems from the fear of being too big and to slow, and not being as flexible as possible; it allowed us to have a working version very early, but now limits somewhat its possibilities. This is why GWM is now deeply redesigned to introduce new general wobs and take advantage of the X11R4 non-rectangular windows in the new version 2 which should be out by march 1990.

5 Status of GWM

GWM has been widely used at the INRIA in Sophia Antipolis, France since June 1988, and BULL has made it widely available by anonymous ftp on `expo.lcs.mit.edu`, with the same

[10] The upper resize handles of **mwm** cannot be emulated

and specialized for other ones, but the user interface designer is free to set up `wd` as felt necessary.

A pattern matching facility, that of the X resource manager, allows to retrieve the `wd` associated with a window from a data base. This pattern matching can be designed such that all instances of the same client (eg all running xterm's) share the same `wd`, and different client classes inherit from the default but add more icons or functions, e.g. a particular menu.

events matches X events (such as `buttonpress`), but with a little processing: The `enter-window` event for instance is **much** simpler than its X11 counterpart in that it tells you only if the pointers enters the bounding box of the wob.

X objects: pixmaps, cursors, events, labels allows to share X resources via the WOOL memory management

3.3 Operation

When you start GWM, it:

- connects to the display it must manage to initialize its X11 structures
- reads and interprets the user's WOOL profile (searched for in a user-defined path) This profile must define at least two WOOL functions:

  describe-window which will be called by GWM to know how to decorate any client window and must return a list of two window descriptors, one for the window itself, and one for the associated icon

  describe-screen which will be called by GWM for each managed screen and must return a window descriptor

  These functions can of course use the X resource manager database facility to perform inheritance on decorations, and decide which decoration to use on any criterion such as the number of windows already on the screen, the size of the new window, etc...

- looks if it is the only window manager running, aborts otherwise
- decorates the managed screens by calling the user-defined `describe-screen` function on each screen (a screen decoration is its background and fsm)
- decorates all already mapped client windows, by calling `describe-window` with the current client window being set to each window, and executes the `opening` function for each window
- executes the (user-defined) `opening` function for the screen
- enters GWM main loop, which consists in:
  - waiting for an X event
  - examining the received event, and if it is for an already created wob, send it to the *fsm* of this wob, else if it is a new window which is mapping for the first time, decorate it (by calling `describe-window`), and execute its `opening` function[9].

[9] the used-defined opening function allows such things as interactive or automatic placement. A pending `closing` function is of course called when the window is destroyed

atoms associates any WOOL object to any other via a hash table. Only one wool object can be referenced, which implies that a `(setq foo 1)` assignment will remove any function definition made by `(defun foo ....` There is **no** limit on atom names length. Atoms names can be composed of any printable ASCII character except `" ' ( )`, and cannot begin by a digit.

active values are predefined atoms that can be used as atoms or functions. If `foo` is an active value, then `foo` and `(foo)` give the same result, and the calls `(setq foo obj)` and `(foo obj)` have the same effect.

Active values are just like other atoms, but setting and evaluating them triggers specific functions to allow for greater power. For instance, just setting a wob's `borderwidth` will actually resize the corresponding X window's border.

numeric variables are atoms that can only store integer values

integers are 32 bit signed integer values

strings are 8 bit strings of characters, with no size limit

functions that can evaluate or not their arguments, and have a fixed or variable arity

fsms the finite state machines. They are WOOL objects in order to be shared among wobs, and respond as well to X events as to WOOL-made events, the so called "user" events.

A syntactic construction has been developed in WOOL in order to specify easily fsm's, using some sort of HyperTalk language. For instance, the syntax to specify that clicking the left mouse button should result in moving the current window.

```
(fsm-make (on (buttonpress left alone) (move-window)))
```

Specifying that the user will start to move a window by clicking in the top bar with the left mouse button is simply achieved by associating the previous fsm with the bar. The reader will notice that it is possible to associate different fsm's to the different bars of a window and then obtain different behaviors for clicks with the same button in the different bars. This would probably not a desired user interface, but it shows the flexibility and the power of the scheme.

A user then programs GWM in a **reactive** way; he codes procedures that will be called on some events, having access to to current wob, window and screen by active values.

wobs descriptors: plug, bar, menu, client are used by GWM to build its wobs.

A key notion is that any window under control of the Window Manager, (remember that X allows to have windows under exclusive control of the application) is managed through a *window descriptor*, noted wd. A wd is a similar notion to that of a Graphic Context in X-window: it is a structure providing control on the operations. There may be as many wd as windows, but the idea is clearly to have a basic default wd imported and specialized by other clients so that all instances of that client (e.g. all running xterm's) share the same wd.

It possible to have either all windows managed in the same style, using a single wd for all of them; or to specify at most as many wd as windows, each specifying a different behavior. For ergonomic user interfaces, it is probably better to have a basic default wd imported

Window A X11 application, such as XTERM(1) or XCLOCK(1) usually opens one or more windows directly on the screen (in fact the *root window*). We will call these windows, which are not created by GWM, **client windows**. GWM will "decorate" these client windows by *reparenting* them, i.e. by creating a **Window** wob and making the client window a son of this newly created window.

A window is a wob made by adding around the client window four (optional) *bars*, and then making a window just enclosing the client window and its bars. Note that the inside of a window wob is thus never visible, since it is always hidden under the sons of the windows that are the client window and the bars.

Bar which is the only extensible wob, and has a width and a length. It is a line (vertical or horizontal) of *plugs* centered on the axis of the bar with optional stretchable space between them.

Plug the simplest of all wobs, its contents is just a graphic which is displayed in their X window. They act thus like some kind of button.

Menu a bar of bars (horizontal menus consists of a horizontal bar of vertical bars, vertical menus are a vertical bar of horizontal ones).

Menus are the only "stand-alone" wobs, their windows are directly created by GWM on the screen. They can be used to implement pop-up, pull-down or fixed menus, control panels, message windows and icon boxes.

3.2 Wool

WOOL is a lisp interpreter of a special kind, since it has some important design differences from "real" lisps:

incremental garbage collection WOOL has a *reference-count* driven memory management scheme. This means that the load of memory management is evenly distributed on all phases of computing, thus avoiding the dreadful garbage collection pauses. But, since reference count memory management doesn't work with circular lists, no WOOL function allows the user to do physical replacements on lists[8].

no real lists, but arrays WOOL lists are internally stored as arrays, speeding up list scanning, and decreasing the space used by lists. We do not really need the generality of the chained cells model, since most lists used in lisp stores program texts, which do not need to be dynamically expanded.

monovalued atoms in classical lisp dialects, you can give a variable and a function the same name without conflict. In WOOL, an atom can only have one value. This saves memory consumed by atoms.

internally typed objects all the WOOL interpreter is written in an object oriented way. This is not visible to the user, since we do not offer a way to define user types, but greatly improves the modularity of the code, and decreases the number of primitives by the use of generic functions.

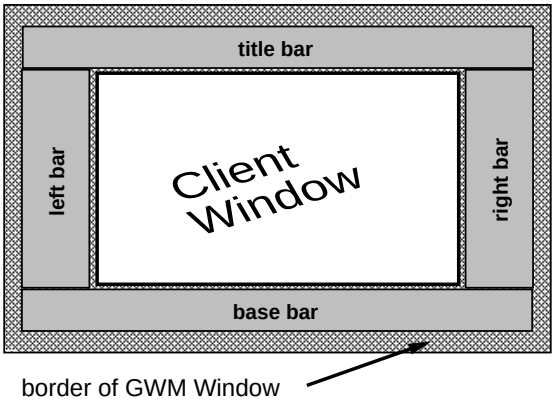
The different WOOL objects are:

[8] In fact, some do, but as flagged as "for experts only" in the documentation

3 GWM technical overview

3.1 Terminology

The standard way used by nearly all Window Managers is to "decorate" a window to be managed with a number of icons, a title, etc. This decoration appears outside of the client window. One should clearly not use the space required by the applications for Window management purpose. Since a window is needed to place this decoration, a new window is created, surrounding the client window. Hence for each window used by an application (and managed by the window manager), there are two windows for the Window Manager, the *inside window* called the **client window** created by the application, and the *outside window*, called **window** created by the window manager which *reparents* the client window to the window, i.e. moves it in the window hierarchy to make it a son of the outside window. The client process totally ignores the existence of the outside window, only the window manager knows about it. A window usually appears to the user as a rectangle surrounded by a border. Since the outside window totally surrounders the inside one, there are four rectangles in the difference. Matching those rectangles is the notion of a *bar*.



WOOL relies on an object-oriented approach. At the root of the object hierarchy is the wob class. WOB is an acronym for Window OBject. To each wob is associated an X Window, similarly to the X-ToolKit[7]. A **wob**, not unlike a X *Widget* is an object composed of:

a X11 window used to display the wob on the screen (output)

a WOOL automaton finite state machine used to trigger **WOOL** functions in response to X11 events sent to the wob (input)

To handle events dispatched to them, wob's have a *finite state machine*. Each wob has its own finite state machine (fsm). When an event happens on a wob, it is filtered by the fsm. This will likely cause the fsm to progress from its current state into another one. When entering an accepting state a table is looked up for call-back routines, which are fired if they exist.

Wobs are not directly created with WOOL constructors, they are **described**, and GWM will use this description to physically create the wobs when it will need to, when decorating a new client window for instance.
Like any X11 window, the user can choose the width, color(s) and tiling pixmap of the border of the wob (which is considered to be in the wob for input purposes). These wobs are of four types:

[7] In fact we designed our own toolkit for performance reasons

2.3 Design choices

In order to reach those goals, an approach was followed for GWM realizing a transposition into the X-Window world of the EMACS[5] editor where new editing functions can be added, and personal keys bound to those functions. EMACS is an editor where basic editing commands are defined as Lisp functions, a specific Lisp that knows about editing and includes a number of primitive functions. New commands can be created by coding new functions, and dynamic behavior may be obtained. The customization of a screen editor is limited to keystrokes, but emacs provide a simple way of redefining the meaning of keystrokes. In the Interpress Emacs version[6] the Lisp language is not a full implementation but a fast, small, reduced implementation of Lisp, called MockLisp, implemented in C, and the whole implementation takes around 256K. This makes emacs usable on most machines with fast response time, yet it allows flexibility and customization.
This efficient and practical scheme has been transposed in GWM to a full X Windows environment. The basic idea was to develop first a Lisp like language, specifically dedicated to window management, using an object oriented approach. It is called WOOL: Window Object Oriented Lisp. WOOL knows about X-Window, has primitive functions for handling windows, and offers a framework for implementing a window management policy.
Implementing a particular Window Manager then amounts to coding a few pages of WOOL code. As with EMACS, however, not everybody will extensively customize GWM, but rather one person per project (such as the system engineer).
The WOOL language has a Lisp syntax and supports a number of Lisp features, however it is clearly not a full Lisp. WOOL is not Common Lisp for X (CLX) because it was felt mandatory that never any interaction would be delayed by garbage collection, that the interpreter should take less than 100 K, and specific window management concepts were necessary. On the other hand, it supports real memory management and parameter passing, unlike Interpress Emacs, where developing a large amount of lisp code is somewhat cumbersome without these features.
One well known problem with Lisp is garbage collection and saturation of the memory space. Because it is unbearable for a Window Manager to garbage collect during interactions, WOOL uses an original technique borrowed to modern ephemeral garbage collectors. WOOL uses a memory management technique distributing the load of garbage collection on every computation, providing a smooth operation, eliminating the need to periodically stop handling user input. There is no specific memory space, so guaranteeing that there will be no deadlock due to memory shortage.
WOOL and GWM are coded in an object-oriented layer on top of C, giving us the necessary power of object oriented programming with nearly no portability problems.
One big drawback to the EMACS approach is that if everybody customizes its environment, you end up with as many incompatible editing environments as there are people under emacs. Thus we want to develop a set of rules, called the *standard profile* which should allow users to share easily decorations and placement algorithms.

[5] *Gnu Emacs* designed by Richard M. Stallman
[6] *Gosling Emacs* designed by James Gosling

2.2 Design goals

When designing GWM (Generic Window Manager) we started from the following principles:

- There does not exist a "best" user interface. It seems that there are different user interfaces well suited to different environments, for instance office world, C.A.D, software engineering, etc. It was felt much too man-power consuming[2] to develop each time a Window Manager appropriate to an environment, and that it would be much more flexible to offer a Generic Window Manager instead, that is, a base tool that can be used later to implement different Window Management policies and different user interfaces.

- When a policy has been defined, it should yet be possible for a final user to customize the user interface with personal preferences, for instance to use left-click instead of right-click to perform a function, or to display custom images in icons. "Power users" want different features than novice users.

- The window manager should have dynamic possibilities. For instance, it should be possible to implement a tiling window placement routine. In order to do that, one needs to execute specific code whenever a window is mapped or unmapped. Besides, every user of a windowing system has noticed that he spends a non-negligible portion of his time tidying his screen desktop by moving icons, arranging window alignments, etc. This is a complex problem which needs to be resolved by innovative algorithms[3] and a way to implement them in the window manager[4].

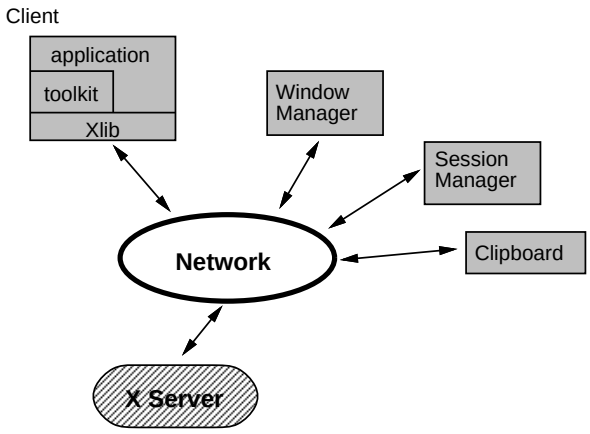
- The window manager should provide to the different clients the possibility to have specific enhancements with respect to window management, and provide an inheritance mechanism. Clients that do not specify any particular property, or buggy clients that do not respect the inter client conventions, should inherit a default so that their windows can be managed properly. But clients that do want a particular way of managing their windows should be able to specify that, and inherit the defaults too. This will lead to an homogeneous user interface, yet not clumsy. For instance one can easily imagine that the management requirements are different for the terminal emulator client (`xterm`) and for the clock.

- No human interaction should be hardwired into the basic code, so that window manager designers (assumed to be assisted by people with knowledge in Human Factors) can experiment and modify very quickly the user interface.

- Yet window management is complex, the Window Manager should rely on a small set of sound concepts with efficient implementation. Interactions should have fast response, and the Window Manager should not use too much memory (typically less than 512 K), so as not to monopolize the resources of the workstation. This is why we choosed not to be **toolkit-based**, trading off ease of programming for more speed, more power, and less space.

[2] And it is really the case if you want to create a fully ICCC-compliant Window Manager now
[3] This is why we want the user to be able to design himself the algorithms, to test his ideas
[4] Thus the need for a powerful extension language for GWM

other managers such as the **clipboard manager** used to keep track of selected items.



The X window system has introduced a precise definition of Window Manager. A X Window Manager is a particular X client, interacting directly with the user for a number of window operations, typically moving, resizing and iconifying a window. The duty of a Window Manager is limited because it is a program which traps a number of X protocol calls such as `XCreateWindow`, `XMapWindow`, but not all calls. Thus it can handle only those cases that it knows about. It is an X policy that any application should be able to run, even though there is no Window Manager running at all. A Window Manager may also communicate explicitly with clients, and this communication is ruled by the X Inter Client Communication Conventions. The application programming interface of an application to a Window Manager is then limited by those conventions, and in any case is achieved through some sort of message passing using window properties (remember that an X Window manager may be running on a different machine than a particular client, maybe with a different file system).
Moreover, as X Windows aimed at replacing every other windowing system in use in the industry, it does not enforce any policy, allowing for anybody to be able to re-implement their previous windowing environment functionnalities on top of X.

2 Why a new Window Manager?

2.1 Existing ones

A number of Window Managers have been provided on the X-Window tapes, notably `wm`, `uwm`, `awm`, `twm`, `rtl`. We were not satisfied with them, because:

- Each of them implemented a particular policy or *look and feel*. e.g. `Rtl` is a tiling window manager, `uwm` has no title bars. That meant there should exists as many window managers as previous windowing environments, and we felt we should try to keep the policy-less aspect of X in one of its window managers.

- They are not programmable, only configurable through a file (e.g .uwmrc for uwm) read at startup containing only static data, typically mouse button assignment to built-in functions, menu descriptions, etc. With such a mechanism, one cannot implement window placement strategies, based on current window locations or sizes, without recompiling them[1].

[1] `rtl` can be compiled with a *neatening* package to provide cleaning algorithms

GWM
the Generic Window Manager

Colas Nahaboo
colas@mirsa.inria.fr
Vania Joloboff
vania@osf.org

BULL Research FRANCE – Sophia Antipolis

November 1989

Abstract

The GWM (Generic Window Manager) is an extensible Window Manager for the X Window System Version 11. It is based upon a WOOL (Window Object Oriented Lisp) kernel, which is an interpreted dialect of Lisp written in C featuring incremental garbage collection via reference counting, lists stored as arrays, and object-oriented internal structure with specific window management primitives. The user builds its own window manager by writing WOOL files to describe objects to be added as decorations around X applications windows, as pop-up menus or as independent windows, and by associating with each object a Finite State Machine triggering WOOL actions on response to X events. GWM should be able to emulate easily and efficiently any other window manager, and play the same role for window managers as EMACS do for text editors.

1 What is a X11 Window Manager?

One of the great merits of the X Window System is to clearly separate the different components of a window system, which was before viewed as a huge black box, into separate tasks managed by separate processes:

the server managing input (events), output (bitblt) and display resources (windows, fonts, colormaps) via requests trough a network (Ethernet).

the Xlib converting graphic orders in requests conforming to the X11 Protocol to be sent to the server over the network.

the toolkit providing an object-oriented abstraction for writing user interfaces of applications, called **client** in the X terminology.

the window manager the user interface for manipulating client windows on the screen and managing the screen space.

the session manager used to start, save, terminate, and restore a full session.