An Infinite-Pane, Zooming User Interface Window Manager and Survey of X Window Managers

Submitted by
Evan Bradley

Computer Science

To
The Honors College
Oakland University

In partial fulfillment of the
requirement to graduate from
The Honors College

Mentor: Serge Kruk, Professor of Mathematics
Department of Mathematics and Statistics
Oakland University

(02/15/2018)

Abstract: This thesis describes a zoomable user interface window manager for the X Window System that aims to provide mechanisms for easily managing a large number of windows. This is motivated in part by greatly increased memory capabilities provided to modern computers as well as the relative stagnation of window managers since the desktop metaphor was first implemented. To address this, a window manager was written that allows the user to zoom over an infinite plane, on which windows may be arbitrarily placed. Taking advantage of the properties emerging from this model, algorithms were written to manage the windows using their associated Euclidean coordinates. Furthermore, a menu system similar to those employed in Oberon and Acme was written to provide the user with the ability to exercise greater control over the window manager. To ensure that it is usable on standard systems for a typical user workflow, it was developed on the X Window System, despite the system's shortcomings. While this is the first window manager developed for the X Window System with the synthesis of these features, it takes inspiration from other window managers, user interfaces, and HCI research. As such, a brief discussion on other research interfaces is included alongside a more extensive survey on X11 window managers, which provide a substantial source for contemporary window management research.

1 Introduction

Early conceptualizations of computer interfaces focused on using the processing power offered by computers to extend the information processing capabilities of humans. "As We May Think," a 1945 essay written by Vannevar Bush to describe a path forward for scientific collaboration after World War II, outlined the idea of the 'memex', which would take the form of a mechanical machine used to quickly find and transmit information [1]. Inspired by this, Douglas Engelbart

outlined his idea for a similar system in his essay, “Augmenting Human Intellect: A Conceptual Framework”, which was put into action for the “Mother of All Demos” at a conference in 1968 [2, 3]. The earliest widely-available computer interface to emerge from these ideas came in the form of the Xerox Alto, which presented users with many of the graphical metaphors that are seen in modern systems. It featured a mouse to control a bitmap interface containing windows with text and graphics, and used its graphical capabilities to provide facilities including a text editor, graphics editor, and file manager [4].

GUIs continued to improve after the Alto, which was directly succeeded by the Xerox Star and Apple Lisa. These split off into the windowing systems and GUI standards that we know today, most popularly Apple’s macOS, Microsoft’s Windows, and the X Window System (abbreviated as simply X) seen on UNIX-like systems.

A window manager is a program that concerns itself with the organization of the user’s documents and applications. They are typically GUI programs that operate on other programs written within an operating system’s GUI framework. More sophisticated window managers will often provide two layers of interaction: functions for manipulating individual windows, and systems for dealing with the relationships between windows. Simple window manipulation often includes operations that move or resize a window, which can then be applied to multiple windows if desired. Window organization often takes the form of virtual desktops or workspaces that provide structure to the layout of windows and facilitate switching between windows.

These functionalities are frequently exposed through keyboard commands or on-screen interfaces, but can also be controlled through computer interfaces to other programs such as by a socket or filesystem. Furthermore, the windows themselves can either be floating or tiled, which changes the manner in which the windows are positioned and sized on the user’s screen. In

window managers that support floating windows, windows are freely placed and sized by the user, typically through a mouse with an on-screen interface. Tiling window managers automatically place and resize windows across the screen, and are generally operated with the user's keyboard in place of using a mouse.

While window managers have seen many improvements in the near-half-century since the Xerox Alto was first introduced, many elements of its interface have been reused in modern window managers. Those on Microsoft Windows and Apple's macOS, the two most popular operating systems on desktop computers, have largely retained the menu-driven, floating window interface the Xerox Alto espoused. Their most significant additions are adding workspaces in Mac OS X 10.5 [5] and Windows 10 [6], alongside smaller window manipulation features such as limited support for tiling. Other systems have seen more substantial innovations, particularly within the X Window System, where more sophisticated layout and control techniques like tiling and more powerful virtual desktop models have been pioneered.

Current research often focuses on task-oriented management techniques and methods of displaying more information on the user's screen. Task-oriented techniques include systems like Taskposé [7], which seek to organize a user's windows into groups that represent the activity of which they are a part. Other systems like WinCuts [8] or Laukkanen's window manager [9] aim to take advantage of increasing display sizes to increase the amount of information a user can reasonably process. Zooming User Interfaces have also seen a small amount of research for spatially representing large amounts of information. This research primarily takes place in consumer products such as Prezi, but has seen developments like Laukkanen's window manager and ETH Zürich's A2 OS [10, 11].

2 Motivation

The primary aim of this thesis is to offer a unified view of the current state of window management research, both academic and non-academic, and to present a window manager that combines the best of these ideas with extensions to them.

Academic window management research has largely ignored the improvements made by research outside academia, which in turn results in window managers or similar utilities that exist inside systems that only make a single improvement on the state-of-the-art, or on research systems that may present numerous improvements, but at the cost of compatibility with popularly-used programs. Examples like WinCuts or Laukkanen's window manager are made for Windows, and while they make notable improvements on it, they are otherwise restricted by the fundamental simplicity of the Windows window manager, and furthermore have not been made easily available for public consumption. Similarly, research systems such as Stack & Tile [12], Rio on Plan 9 [13], or ETH Zürich's A2, while presenting a number of novel features, are restricted by their sole availability on obscure operating systems in that they cannot be used by the majority of users who use more complex applications which do not have a port. None of the three operating systems just mentioned have a web browser that works with the majority of modern websites, for example. While these are complete systems and "power users" could, if desired, use them as a primary system, the lack of widespread support limits the effects of the ideas presented in these systems, despite the portability of the ideas across window display systems.

Conversely, window managers available for public use, whether commercially or freely available, have failed to incorporate many of the more advanced features provided by academic window managers. The default window managers available on Microsoft Windows and macOS

only offer a basic array of window management utilities, and are overall considerably behind free offerings. While some extensions have been written to extend their capabilities, these programs are often limited due to the restricted nature of the platform for which they were written.

Freely-available window managers compatible with commonly-used systems, which are primarily Linux-based, but also include BSD systems and a few other UNIX-like systems, have seen substantially more development. This has primarily taken place on X, which has over 200 window managers written with the X11 protocol. These window managers have presented a number of window management interactions and related software architecture advances that have not seen intensive study in academic research. Most notably, the use of tiling has provided a number of novel interaction and organization techniques for handling windows, and the flexibility provided by these systems provides a good base for further exploration by both users and researchers. However, despite the improvements they have made, development has almost entirely been performed in complete separation from academic research. Of the few visible interactions between X11-based window managers and academic research are 9wm's [14] development out of Rob Pike's research that cumulated into 8½ [15] and Rio, and the partial inspiration of Stack & Tile from Ion [16].

Owing to the improvements they have offered against the state-of-the-art in academic research and to the lack of an academic review of window managers built on X, part of this thesis has been devoted to providing a survey of these window managers. The goal of this survey is to examine the different families of window managers based on the lineage of their code and ideas, and to examine novel features that are offered by more advanced window managers. While some catalogs have been made of X window managers, these focus more so on the raw set of features provided by these window managers and their installation and use on a GNU/Linux-based system.

Instead, the window management capabilities and user-interface elements provided by these window managers will be examined in the survey presented in this thesis. Furthermore, we look to examine the current landscape of non-X11-based window managers to assess ideas presented on less accessible, but still publicly-available, window managers.

The culmination of this research lies in a new window manager described in this paper, which was written to bridge the gap between academic and non-academic window manager research, and to extend the state-of-the-art with a few new features. It makes use of a Zoomable User Interface, which has been extensively researched and most recently attempted in Benjamin Bederson's work academically, but has also been described in Jef Raskin's *The Humane Interface* [17], and consequently implemented in A2. Similarly, ideas from Oberon and Acme have been implemented to add to the functionality of the Zoomable User Interface. Finally, the large desktop functionality allows for the encoding of Cartesian coordinates within the windows, and consequently these can be used for a number of novel applications. This thesis argues that the combination of these features will allow for the management of a larger number of windows through creating better mental links between windows in the minds of users, and to allow for easier organization even when not all windows are present on the screen. It is hoped that these will provide a basis for further research on these ideas in an environment that is easily accessible and allows for testing with typical workflows.

While the details of HCI can often become intertwined between the on-screen and functional interfaces provided to a user, this thesis primarily focuses on functionality related to window management rather than on user-centric models of interaction. As such, it is guided by, but does not optimize for, user interface ease-of-use, speed, accessibility, and aesthetics. To ensure a narrow scope and to optimize for compatibility with current systems, it also does not aim to take

a critical look at the desktop or “windows, icons, menus, pointer” metaphors. Instead, it merely aims to extend a currently available system to examine new ideas in a practical and usable manner.

3 X11 Window Managers

There have been a few serious efforts to catalog X11 window managers, but they usually focus on user-facing details like the interface or installation and usability, and are conducted in an informal, non-academic fashion. The Arch Wiki’s catalog on window managers provides a list of window managers and their associated features, with pages on certain window managers describing more in-depth usage [18, 19]. Wikipedia provides a similar catalog, with a greater emphasis on comparison on the basic details of each window manager [20]. More in-depth surveys also exist, such as the Wikibooks Guide to X11 Window Managers, which lists a significant number of window managers with details on installation and features offered by select window managers [21]. Giles Orr has offered the most complete writings on X11-based window managers, including a detailed writeup of his experiences with each of a selection of window managers, and a consistently updated exhaustive list of X11-based window managers including minor details about them [22, 23].

Here, we look at the window managers not just as interfaces, but as systems for managing windows; this includes the code and concepts as well as the interactions the window manager provides. Additionally, the lineage of ideas and code that influenced a window manager are discussed to give a background to the window management ideas that are prevalent today. To provide a complete picture, window managers without any significant features or details are still given mention as a part the lineage of a window manager or window management concept.

Window managers are divided into families organized by their direct ancestry or by a defining feature, if the window manager has a more complex history or marked a shift in philosophy from its predecessors.

3.1 Overview

The X Window System has been discussed in great detail [24, 25] elsewhere, so here we merely give a brief overview of the architecture, libraries, and methodologies used in programming window managers with the X11 protocol. The X Window System is written in a fully network-transparent manner, using the X server that communicates directly with the computer's graphics card to draw graphics to the screen and with any I/O devices for user interaction. Furthermore, it holds its own state about currently open windows and their corresponding information. The X11 protocol is then used to communicate with the X server by client programs, most often window managers, which send commands and receive information from the X server regarding the state of the system. Information received from the X server can include the display geometry, information about currently open windows, and events received from I/O devices.

As the predominant language for system-side development on Linux, C is the most frequently used language in X11 development, closely followed by C++, but as it is merely a protocol, other languages have found use. The oldest, and still most-used client-side library for X11 is Xlib. It allows for full communication with the X server over a socket and can be used to interface with the X server using C or C++. Bindings also exist for other languages, including Python, Common Lisp, Go, and JavaScript.

A newer library, the X protocol C-language Binding (XCB), has gained popularity as

X-based programs seek to modernize their code. XCB is much lower-level than Xlib and only consists of asynchronous calls to the X server, whereas Xlib contains both synchronous and asynchronous calls. It is written using an XML protocol specification, which is then translated by a parser into code in a particular language. Parsers exist for C, Python, Perl, and Emacs Lisp.

Window managers use these libraries to move and resize windows, as well as controlling which windows appear on the screen, which can be applied to both floating and tiled windows. Floating window managers frequently support maximizing a window, where it is resized to cover the screen, minimizing, where it is hidden, and resizing or moving through moving the mouse after activating that functionality. This is also frequently called "stacking," but here we use the term floating to indicate the way the windows are placed. Stacking refers to the ordering of windows by "stacking" them upon one another, but this describes functionality used in switching windows, and does not explicitly refer to how they are placed.

Tiling window managers will frequently provide functionality to automatically resize a window, but will also allow for manual resizing of a window's allotted space, which will frequently result in the resizing of other windows in accordance with the changes to optimize for available screen-space. Windows are frequently hidden in tiling window managers through virtual desktops, but windows can also be layered over one another. Virtual desktops, also called workspaces, assign a label to a group of windows, where a user may then select a label and view the windows corresponding to that label. Windows not associated with the corresponding label are consequently hidden. Both forms of interaction will provide functionality to close a window, and often to open a new window by spawning a new process.

3.2 Tab Window Manager

The Tab Window Manager (TWM), originally named "Tom's Window Manager," was started by Tom LaStrange in 1987 to improve upon what he saw as deficiencies in the Ultrix Window Manager [26, 27]. It was written as a floating window manager that reparents windows to give them decorations, which were used to hold buttons and perform operations on the window. It was primarily controlled through the mouse, and offered a configuration file to change interactions [28]. TWM is written in C, the only language with an X11 interface at the time, using Xlib. Owing to its features, it measured at roughly 11,000 lines of code as of its release for X11R6 [29].

Ideas from TWM have continued on to newer window managers both through its code and concepts. Direct descendents include Claude's Tab Window Manager written by Claude Lecommandeur (CTWM) [30], the Virtual Tab Window Manager by Dave Edmonson (VTWM) [31, 32], and Tom's Virtual Tab Window Manager by Tom LaStrange (TVTWM) [31, 33]. All three have the primary aim of adding virtual desktops to TWM, but have added other features as well, such as a pager in the case of VTWM [32]. LaStrange also developed the Solbourne Window Manager (swm), which built off of TWM's ideas, and added the concept of a virtual desktop that spanned the entire X root window, using the screen as a viewport into that window [34]. It was released in 1990, and was one of the first, if not the first window manager to do this on X; the next closest window managers are VTWM, ostensibly released in 1992 [35] and CTWM, first released in 1992. SWM and VTWM both offer the ability to pan over the virtual desktop, which has a maximum size of 32767 by 32767 pixels, the maximum dimensions of an X window. Windows can be made sticky in SWM, retaining their position on the screen, which by default applies to the pager, which shows an overview of the virtual desktop and provides

interactivity for manipulating it [34]. These features were later copied by Edmonson for use in VTWM, and later by LaStrange for inclusion in TVTWM [31]. Virtual desktops quickly became a popular feature, and are now included in the majority of available X11 window managers.

3.3 NeXTSTEP

NeXSTEP was the operating system written by NeXT, Inc. to run on NeXT computers, which came with a desktop environment that quickly became and has since remained a favorite among those who used it. Users could manipulate windows using the titlebars on the windows and with the dock on the side of the screen. Further interactivity was performed through a menu summoned by clicking on the desktop. Many of these features would find themselves a part of Mac OS X after Apple acquired NeXT in 1996. Similarly, projects like WindowMaker [36] and AfterStep would attempt to recreate the NeXTSTEP environment in a free and open source system.

WindowMaker provides an environment that is very close to the original NeXTSTEP interface, matching its design and functionality as part of the GNUStep project. At around 38,000 lines of C code, it contains a lot of functionality typical of a more complete desktop environment [37]. The desktop menu allows for numerous configurability options, and there are UI elements in the form of widgets, menus, docks, and window title bars that provide the user functionality [38].

WindowMaker also provided an inspiration for the design of Blackbox [39], which can be seen in its “slit”, which provides dock functionality, window title bars, and interaction through clicking on the desktop. Blackbox has spawned a number of derivatives, including Fluxbox [40], which aims to include more features by default than Blackbox, and Openbox [41], which offers a number of configuration and theme options.

3.4 Rio

Rio is the default window manager on Plan 9 from Bell Labs, which was written by Rob Pike as a rewrite of 8½ with a strong focus on concurrency [13], while still maintaining the Unix philosophy used in 8½, as the two are largely functionally equivalent. Rio effectively acts as a multiplexer for the screen, keyboard, and mouse, as it simply reads from the file created for each device by Plan 9. Window functions are all performed through the mouse, which are accessed through a click on the desktop. Windows are created by outlining the window on the desktop for a new terminal, after which all windows created thereafter are contained within the outline of their parent terminal [42, 43]. These ideas were faithfully transcribed into 9wm, a clone of 8½ for X11 written by David Hogan in 1994 in roughly 2,600 lines of C code [44]. 9wm became the basis, either in code or through ideas, for other early X11 floating window managers: AEWM [45], Larswm [46], lwm [47], w9wm [48], and windowlab [49]. The most notable of these are Larswm and AEWM.

Larswm introduced controlling windows tiling and included a status bar by default at the bottom of the screen. The tiling followed a basic stacking tiling model, where a master window is placed in its own area on one side of the screen, while all other windows are sized to be placed in the remaining screen area. It also included a document view mode, where a selected window would be resized with the same proportions as an 8½ inch by 11 inch sheet of paper. The status bar shows the current desktop and contains a string that encodes information about that desktop, and provides an area to click and view a menu for controlling the window manager [46].

AEWM was written by Decklin Foster in 1998 to optimize for Fitt's Law by offering large UI elements, such as a large title bar, buttons, and the user's desktop, for manipulating windows [45]. It is written with approximately 2,600 lines of C code, which has made it a popular basis for

other window managers [50].

Evilwm [51] is the most popular of the AEWM derivatives, offering both usability and functionality extensions as part of its fork. It is written by Ciaran Anscomb, who began writing it in 1999. Like AEWM, it is written in C, and only encompasses 3,000 lines of code (roughly 400 more than AEWM), indicating it is more a refinement of AEWM than a total rewrite. evilwm contains basic virtual desktop support, mouse control, the ability to move and resize windows with the keyboard, and a basic configuration file. It also contains a feature that allows easier window alignment called "snap-to-border," where positing a window within a certain number of pixels from the monitor border or the border of another window will cause the window to move such that there is no space between the two borders. mcwm [52] and 2bwm [53] have been forked from Evilwm and also function as refinements of EvilWM's functionality.

The Calm Window Manager (commonly known as CWM) was originally based off EvilWM, but has since been rewritten from scratch [54]. It is part of OpenBSD, and has been written by Marius Aamodt Eriksen since 2004 [54, 55]. Measuring at 5,300 lines of C, it is the most serious deviation from EvilWM in functionality [54]. It shares most of its base functionality with Evilwm, but includes a more flexible group model allowing windows to belong to multiple virtual desktops, and has the ability to search for windows. Its search feature is its most novel, and allows the user to find a window by searching for its label, current title, five previous titles, or class [56]. This allows locating a window without the requirement to remember where it was placed, with the tradeoff being the number of keystrokes necessary to access it.

3.5 Suckless

Suckless is a software group with a focus on simple software that favors power users. Members of suckless have released user-facing software programs for Linux and other Unix-like operating systems that keep to this philosophy, generally written in C. Anslem R. Garbe, one member of Suckless, has been the initial creator and primary contributor of the window managers WMI, WMII, and DWM, which have been released through Suckless.

WMI was a tiling window manager that was written by Garbe from 2003-2009 [57]. It was designed to be a window manager that combined what he perceived as the best parts of Larswm, Ion, TrsWM [58], EvilWM, and Ratpoison [59] operated through a vi-style interface [60]. WMI had numerous features and was highly configurable, written in around 13,000 lines of C++ code [57].

Window Manager Improved 2, often shortened to WMII, was forked from wmi by Garbe, who wrote for it from 2004-2006, before its maintenance by Kris Maglione from 2006-2014. Measuring in at 11,500 lines of C code, it represents a significant departure from wmi in its design. It took significant inspiration from Plan 9, mostly through its integration of 9p, the Plan 9 network-transparent filesystem interface, and also through acme, which offers tiling window functionality to organize editing panes. Configuration and control of the window manager are primarily done through a 9p interface that serves a virtual filesystem with writable files for available options and functionality [61].

Seeing WMII as being consumed by its own feature-set, Garbe sought to design a highly minimal window manager that would meet his needs, which was initally called gridwm before being renamed to the Dynamic Window Manager (DWM) [62]. DWM has since become the

premiere tiling window manager supported by Suckless. It was first written by Garbe in 2006, and has a 2000-SLOC limit to enforce minimalism as a design philosophy; it currently measures at 2,400 lines of C code [62, 63]. All configuration to the window manager is done through its source code, either through a header file or by directly amending DWM's primary source file. Frequently, these changes are distributed in the form of patch files [64]. Stacking and maximized tiling modes are supported in addition to a floating mode. A bar is provided to display information and select between different workspaces, which are defined as tags in DWM. The primary difference between DWM's tags and normal workspaces are that a window can be assigned to multiple tags, and if desired, multiple tags can be displayed at once [63].

Owing to its small code base, DWM has become the basis for other tiling window managers, both through ideas and code. CatWM [65], dminiwm [66], monsterwm [67, 68], and FrankenWM [69] are direct descendents, which aim to be more focused versions of DWM, and include goals such as smaller codebases or more tiling layouts. Awesome was also initially written as a DWM fork [70] with the intention of removing the SLOC restriction and incorporating feature patches to DWM. It was started in 2007 by Julien Danjou (now maintained by Uli Schlachter), and has grown to roughly 9,000 lines of code [71]. 2wm [72], Xmonad [**xmona**], and i3 [73] have also taken inspiration from DWM.

3.6 Scripting

Scripting has become a popular feature in window managers as a method of allowing users to easily add functionality. In particular, many window managers use languages such as Lua to provide a user with a popular, Turing- complete, and well-supported language to script their

window manager.

One of the earlier window managers to do this is Ion, which began development in 1999 and continued development until 2009 [16]. Ion supports basic static tiling functionality, and later became known for its tabbed layout, which allows the user to switch between frames on a workspace. These frames could themselves be divided into sub-frames. Ion is a substantial window manager, registering at roughly 44,000 lines of C code and 4,000 lines of Lua code [16]. This is in part due to Ion's integration of a Lua interpreter, which is used to script any part of the window manager. Awesome also supports a similar Lua interface for control.

Another early window manager to offer scripting capabilities is Sawfish, a floating window manager first developed in 1999, and sees current development [74, 75]. Sawfish is scripted using rep, a Lisp-like scripting language that comes with similar facilities to regular Lisps, like an Emacs interaction mode and REPL [74]. Sawfish also supports "large desktop", and comes with a pager to navigate this desktop by panning the actual screen as a viewport over a number of screen-sized "cells", for which there can be many [76].

Qtile [77] and PyWM [78] are both scriptable in Python, which is more mainstream than rep or Lua. PyWM was originally written in 2003 by David McNab (continued in 2006 by Elmo Mäntynen), exists largely as a small (1,000 lines) wrapper around FLWM, which is thereby scriptable in Python [78, 79]. Qtile represents a more extensive effort, with a code base amounting to roughly 17,000 lines of Python, an extensive test suite, documentation, and interaction methods through a shell [80, 81]. Since it is written in Python, it can be scripted using the same language since Python can be interpreted.

Xmonad [82] is analogous to Qtile for its use of a single language for both the core window manager and for scripting. Xmonad was started by Spencer Janssen in 2007 as a window

manager written in a purely functional style using Haskell [83]. It has also been formally verified through a rewrite in Coq by Wouter Swierstra, which passed the Xmonad test suite and fulfilled the functionality covered by these tests, showing the rewritten portions of code to be essentially bug-free [84]. Users may simply use Xmonad's API to add interaction and tiling techniques. It is relatively simple at its core, comprising 1,700 lines of Haskell. Extra functionality is included in the Xmonad-contrib package, which contains hundreds of extensions. These include actions that interact with some aspect of Xmonad, configuration options and utilities offering interactions or customization, hooks for running code after events, and tiling layouts.

StumpWM [85] also follows this philosophy through its use of Common Lisp. StumpWM was started as a reimplementation of Ratpoison by Shawn Betts, and is now maintained by David Bjergaard [86]. It runs inside a compiled Lisp image that allows for on-the-fly reconfiguration of both extensions and window manager code, which lends itself to StumpWM's attempt to be the "Emacs of WMs" [86]. StumpWM's code is defined within a Common Lisp package that may be accessed during runtime, but is written to expose an API to be used by any user-written code; it consists of roughly 12,000 lines of Common Lisp code [86]. The flexibility of StumpWM's runtime assist in features that include commands that can accept user input and are executed by the window manager, hooks that allow access to X events, and an input bar that offers the ability to run and edit Lisp code in a miniature REPL. Configuration of StumpWM is also done through Lisp, offering the same level of control as pre-compiled code. StumpWM offers many of the same basic features as Ratpoison, but sees more development, and offers the ability to extend it through user-added code. Features considered not general enough for inclusion with the main distribution have been developed as modules, which extend StumpWM through its API. These include utilities that interface with or extend the functionality of X or the operating system, interact with a

particular program, or display information on the modeline [87].

Taking inspiration from StumpWM and TinyWM [88], the Common Lisp FullScreen Window Manager (CLFSWM) was developed by Philippe Brochard starting in 2005 [89, 90]. It was written using the same CLX backend as StumpWM, but with a completely separate code base, and currently comprises around 11,600 lines of code [90]. CLFSWM organizes windows using frames organized in a tree-like fashion, which then can be navigated to use a certain window. The X root window functions as the root of the tree, within which other frames and windows can be placed. These frames may be switched to in a virtual desktop-like manner by maximizing them and making them the current root, where windows and other frames may be placed and navigated to. Each frame can have any number of children frames or windows, and children frames are visible even when a parent frame has been selected as the current root frame. CLFSWM also allows flexibility within this scheme: frames can apply tiling layouts to windows within themselves, windows can be in more than one frame, and rules can be applied to windows for placement on the screen and which frames they belong to [89]. Like StumpWM, CLFSWM can also be controlled and extended through Lisp.

Scripting offers far greater flexibility than can be afforded by window managers that simply take configuration options, while allowing a cleaner interface than directly editing the source code of a window manager. In essence, it allows for the user to create new management techniques and interactions on top of the window manager, using its core as a base. This can improve the portability, stability, and development speed of additions by removing any direct interfacing with the X server from the application.

A key feature of some window managers is simply the language they're written in, since the window manager may then be controlled and configured through that language. To take

advantage of this property, these window managers will often structure their code in such a way that the internal code itself is sufficiently modular to be modified, or that it is well-encapulsated and offers an API suitable for extension through modules.

3.7 Socket-control

To free users from having to use a particular language when scripting their window manager, other window managers provide socket interfaces that can be used by any language. In this model, the window manager acts as an intermediary between the client program and the X server, combined with some of its own rules or functionality.

WMII was one of the earliest window managers to offer this functionality, which it implemented through a 9p interface [61]. Since 9p is network-transparent, any program capable of networking with WMII can simply connect to its interface, mount the virtual filesystem it presents, and write to predetermined files to interact with it. Despite the flexibility offered by this interface, it has not been used in any of its successors, which have used local sockets or simpler socket protocols.

Inspired by some of the ideas from WMII, but having a different vision, Michael Stapelberg began developing i3 in 2009 [73, 91], and it now contains a host of features, written over 16,000 lines of C code using XCB. i3 extends from WMII's dynamic window management strategy, automatically placing windows and organizing them in a tree structure. The tree layout of i3 uses the X root window as the root of the tree, and creates two frames from it, split either horizontally or vertically. Each frame can contain either a window or another pair of frames. The tree can be as large as the user wants, or as large as required to fit the user's windows. Each virtual

desktop contains its own tree holding the frames and windows [92]. i3 also can be controlled through a Unix socket it opens, and the `i3-msg` utility can communicate with it through this socket. Through this interface, `i3-msg` can send any command available in i3's configuration file for use in keybindings, or can issue commands to retrieve information from i3 [93].

Herbstluftwm takes inspiration from both i3 and WMII, as well as Musca and Xmonad, focusing on a more socket-driven interface [94]. Thorsten Wißmann began developing it in 2011, and today it comprises approximately 12,000 lines of C++ code using Xlib [95]. Herbstluftwm is controlled entirely through its Unix socket interface, herbstclient, which is invoked as an executable by another program and passed arguments to send to herbsluftwm [96].

This approach has also been adopted by the Binary Space Partitioning Window Manager (bspwm), written in about 10,000 lines of C code since 2012 by Bastien Dejean, which uses Dejean's programs `bspc` and the Simple X HotKey Daemon (sxhkd) [97] to control bspwm [98]. Binary Space Partitioning works in bspwm in a fashion similar to i3, where the X root window is split into pairs of frames, which themselves may hold a window or two more frames. Only the replace and pair functions may be applied to the nodes of the tree. The replace function replaces a node with a new one and moves the old node to another branch, while the pair function replaces the node with a new parent node and places both the old and new nodes as children inside the new parent node. To control bspwm, `bspc` is invoked in a fashion very similar to herbstclient, and this is often used in tandem with sxhkd to bind keyboard and mouse inputs to bspc commands [98].

Basedwm also can be controlled through SXHKD, though it is a significant departure in ideology from the previous window managers. BasedWM is a markedly simple floating window manager, written by Antti Korpi from 2014 to 2015, consisting of roughly 400 lines of LiveScript, a language that compiles to JavaScript [99, 100]. BasedWM offers the ability to pan the desktop

by moving all currently mapped windows by a specified distance. The effect of this is the appearance of moving "across" the desktop, when in fact it is the windows that are moving [99]. The secondary consequence of this is that the desktop is essentially infinite, since windows can be panned out of the viewport offered by the new screen, allowing for more windows to be placed on the newly-empty space. This model is also not restricted to grids sized to the screen like in other window managers such as VTWM, offering greater flexibility, but also less support for alignment. To take advantage of the large desktop, Korpi also wrote Hudkit, a transparent web browser that shows the windows on the user's desktop, that gives an overview of the desktop much like a pager [101].

3.8 Unconventional Window Managers

Some window managers seek to provide users with more customizable or tailored interactions through appealing to a niche. The Emacs X Window Manager (EXWM), written by Chris Feng [102], is an example of this as a window manager written for use in Emacs. EXWM uses the X protocol Emacs Lisp Binding (XELB) as its basis, which converts the XCB specifications to Emacs Lisp directly from XML. It is then loaded as an Emacs package and stores windows as Emacs buffers, which can be operated as normal. EXWM also includes functionality for workspaces, floating windows, multi-monitor, compositing, and a system tray [103]. Pyro Desktop took a similar approach to EXWM by placing window management facilities in Firefox, using Firefox's (now deprecated) XML User Interface Language (XUL) in combination with X compositing to accomplish this.

It is also possible to avoid using a proper window manager at all, instead controlling

windows through tools that separately communicate with X. No-WM [104] claims that the functionality encompassed by most window managers is orthogonal (i.e. inherently disjointed), it should be dispersed into separate programs. To this degree, it only contains basic facilities for positioning and switching between windows, totaling about 400 lines of C code to accomplish this. For any other functionality, such as binding keys, launching programs, or other utilities, it refers to other tools. WMutils [105] takes a similar approach, except with the addition of more tools, with the core programs clocking in at roughly 900 lines of C code [106], written on top of XCB. It provides additional tools, such as the ability to view information about windows, change their border, and moving the pointer. The different components of WMutils are intended to be tied together through scripts to compose a full window manager, as shown in the `contrib` repository [107].

3.9 Discussion

New window managers written for X will often draw inspiration from particular attributes of other window managers in their development. The open-source and highly-standardized nature of the Unix-like ecosystem has contributed in large part to the evolution of the field, as developers can draw on previous research more directly than in an academic context. To illustrate the current state of research, 11 window managers were selected for display in tables listing relevant attributes. These were selected for their relative popularity or influence, feature set, and uniqueness compared to other window managers.

While many X window managers were developed since the release of the X Window System in the late 1980's and leading into the 2000's, the latter half of the 2000's onward has seen

an uptick in development: 7 of the 11 window managers selected began development after 2005. This is likely due to increased internet access, and the popularity of code-sharing websites such as SourceForge and GitHub, which currently or have previously hosted many of these window managers. It is important to note, in addition, that the level of activity of a window manager will often impact its development and influence; most of the window managers listed below are in relatively active development. The exceptions to this are FVWM and EvilWM, which offset the need for further updates with the length of time they have been in development and the completeness of their feature set relative to their goals.

Name	Initial Release
Awesome	2007-09-05
BSPWM	2012-07-28
DWM	2006-07-10
EvilWM	1999
EXWM	2015-07-17
FVWM	1993
i3	2009-02-06
Openbox	2002-04-11
StumpWM	2003-07-21
WMutils' core	2014-11-26
Xmonad	2007-03-07

Table 1: Source code attributes for chosen window managers.

The source code of a window manager provides a method to directly continue research, as it allows research to continue with minimal setup costs. The two primary factors involved with this is the number of source lines of code and the language used to develop the window manager. Window managers that contain only a few thousand lines of code tend to see more forks than counterparts with substantially more. This has two likely primary contributing factors: smaller codebases are easier to work with, and smaller window managers are likely to provide a more minimal base of ideas that can be extended without extra features. 9wm, AEWM, and DWM are all examples of this: each is composed of roughly 2,500 lines of code, and has numerous forks.

Newer window managers tend to use XCB instead of Xlib, though not universally. XCB is typically seen as a leaner, more straightforward, and faster alternative to Xlib since it is composed of entirely asynchronous calls. Some window managers like Awesome or FrankenWM, both based on DWM, have either rewritten DWM routines or written new routines to adopt XCB, or in the case of FVWM, have declared an intention to replace Xlib calls with XCB equivalents [108].

Name	Language	Approximate SLOC	Library
Awesome	C	13,700	XCB
BSPWM	C	10,000	XCB
DWM	C	2,400	Xlib
EvilWM	C	3,000	Xlib
EXWM	Emacs Lisp	6,500	XELB
FVWM	C	148,900	Xlib
i3	C	16,000	XCB

Openbox	C	34,500	Xlib
StumpWM	Common Lisp	12,400	CLX
WMutils' core	C	800	XCB
Xmonad	Haskell	1,700	Xlib

Table 2: Source code attributes for chosen window managers.

Dynamic window management has become the default tiling philosophy in the majority of newly-released tiling window managers for its flexibility and the numerous options available in tiling algorithms. These algorithms can be changed for a workspace at runtime, reorganizing the windows, and can in some cases be nested or selected per-workspace. The following is a list of some popular layouts. Note that the names of layouts differ between window managers, and many offer both horizontal and vertical flavors.

- **Maximized:** A single window covers the entire monitor. Windows are either switched through keyboard bindings, or through on-screen tabs (typically called a tabbed layout).
- **Grid:** Windows are placed in columns, rows, or cells based on specified grid dimensions.
- **Tree/Binary Space Partitioning:** The focused window or frame is split on the creation of each new window, forming a tree of windows. Binary Space Partitioning is a specialization of this and splits the window into two even sections.
- **Stack:** One or more windows fill a reserved master area, while all other windows are

arranged in a specified pattern inside one or multiple stack areas. This is typically done in a pattern resembling a fibonacci spiral.

Floating windows are allowed in tiling window managers as well, which have modes for handling dialog windows created by programs or windows that do not conform well to tiling. These either come in the form of a floating layer above tiling windows, or a separate floating mode where tiling is disabled. As tiling is a subset of floating, in that it simply automatically calculates and places windows, instead of putting that responsibility on the user, it is technically the default state of a window manager. This is seen in WMutils, which despite not technically being a window manager, provides utilities for a user to implement both schemes with minimal friction.

Name	Type	Graphics
Awesome	Dynamic	Info bar, menus
BSPWM	Dynamic	–
DWM	Dynamic	Info bar
EvilWM	Floating	–
EXWM	Manual	Through Emacs
FVWM	Floating	Titlebar, menus, taskbar
i3	Dynamic	Title bar, Info bar
Openbox	Floating	Title bars, menus, taskbar
StumpWM	Manual	Input/output line, modeline
WMutils' core	–	–
Xmonad	Dynamic	–

Table 3: List of interaction attributes for chosen window managers

4 Other Systems

4.1 Windows and macOS

Despite the plethora of X11-based window managers available, Linux and other Unix-like operating systems only comprise an estimated 2.33% of the desktop operating system market [109], where X11 is primarily used. Another 96.93% of the market goes to Windows and macOS, which have an estimated market share of 89.01% and 7.92% respectively. Both operating systems have remained largely faithful to the vision of the desktop laid out by Apple's early attempts with the Apple Lisa and Macintosh. Each uses floating window management as its primary style, with minor tiling features included. Window decorations offer the primary source of interaction, with some keyboard shortcuts allotted for specific actions. Native configurability compared to previously discussed window managers is low, with few options being given to stylistic or functional customization. Owing to the popularity of these windowing systems and the narrow scope of their feature set, we explicitly enumerate their capabilities.

Windows offers simple stacking capabilities, operated through the window's title bar decoration and the start bar, with corresponding keyboard shortcuts for most actions. The window decoration presents three buttons: one to remove the window from the screen, one to resize it to

encompass the entire screen, and a third to close the window. Users are also given basic tiling functions by dragging a window to the left or right sides of the screen. If desired, all windows can be hidden through a button on the start bar. Basic virtual desktop functionality was added in Windows 10, allowing a user to add a new desktop, add windows to that desktop, and either switch to or remove that desktop. A menu is also offered to switch between windows, or, in Windows 10, a user can enter a view that displays all windows adjacent to one another, and raise and focus a window from those.

macOS offers slightly more advanced capabilities than Windows, but still shares the majority of its feature set. When resizing a window to fill the screen, it may either be resized within the current virtual desktop, to the limits of the menu bar and dock, or may be placed within its own virtual desktop in a typical full-screen fashion. Windows un-maximized from their own workspace will return to their original workspace. Two windows may be vertically tiled within these full-screen virtual desktops, and the ratio of their sizes may be resized by dragging along the border separating them.

4.2 Wayland

Wayland [110] is a protocol that is intended to serve as a replacement to X. X was originally developed in 1984 as a network-transparent display system with the intention of being run and displayed on different computers, one server and one client, respectively. This model is considered outdated due to the current model of computing, where computers are sufficiently powerful to render and display graphics in a single machine. Similarly, the complexity of the X codebase has drawn criticism for its complexity and difficulty in maintenance. While Wayland

has been in development since 2008, it is still used by a minority of users of Unix-like desktop operating systems.

Sway [111] is an early popular Wayland compositor and window manager written by Drew DeVault et al. which seeks to closely replicate the functionality of i3. As of writing, it is compatible with the majority of i3 features and commands, including i3's configuration file format [112]. The Wayland protocol moves complexity that would normally be in the X server into compositors however, resulting in Sway's codebase measuring at roughly 21,000 lines of C code, despite being roughly equivalent in features to i3, which is close to 16,000 lines.

Way Cooler [113] is a similar Wayland compositor and window manager written in Rust, which takes inspiration from both i3 and Awesome. As such, it is scriptable in Lua and has planned compatibility for Awesome. Way Cooler is written by Preston Carpenter et al. and consists of approximately 13,000 lines of Rust code [114].

Currently Sway and Way Cooler are both built on the Wayland Compositor [114, 115] and its respective Rust port. However, both have plans to respectively move to C and Rust versions of wlroots, a new Wayland Compositor written by the authors of Sway and Way Cooler for providing basic utilities for Wayland Compositors [116] in a fashion similar to Xlib or XCB.

4.3 Experimental Operating Systems

Experimental and research operating systems have the advantage over more established systems of incorporating more advanced features deeper into their architectures, allowing greater sophistication in higher-level applications. This is not always a product of the core operating systems so much as the philosophies that guide the development of these operating systems, and

the changes enabling more experimental UIs often reside in the display servers, which generally can be operating system-agnostic. However, these changes frequently are seen in the desktop environment of the operating system for which they are built, due to the self-contained nature of most experimental operating systems, which requires that they recreate their own versions of everyday tools.

Haiku is an operating system that aims to recreate BeOS, a now-defunct operating system created by Be, Inc. Haiku offers a unique interface that noticeably differs from most other environments. It uses TWM-style window decorations that only cover a portion of the top of a window's upper border, which aptly allows for windows to be stacked in a fashion similar to the tabbed features offered by Ion. Windows can also be tiled by attaching their borders, which links them when one is moved or resized. Similarly, windows can be moved across up to 32 workspaces, divided into a maximum of 16 columns and 2 rows. Each workspace is highly independent: moving a widget in one workspace will not cause it to be moved in others, and each workspace can have its own monitor settings. Haiku uses the deskbar as the primary menu, which offers menus for starting an application and lists all currently-open windows much like the TWM icons menu. A study by Zeidler et al. showed that the Stack & Tile features they wrote, which perform the aforementioned tabbing and tiling, improved user effectiveness in managing windows, specifically with task completion times and interface satisfaction [12].

A2, formerly AOS or Bluebottle, is a new version of the Oberon operating system based on the Active Oberon language. A2 uses the standard minimize, maximize, and close buttons within a top-edge window decoration to control windows. Windows are all floating, and can also be resized by dragging along the window's borders. It supports an infinite large desktop, and allows windows to be placed freely within this plane, even to be resized outside the boundaries of

the screen. The desktop can also be panned by moving the mouse to the edge of the screen and holding it there until the desired coordinates are reached. A2's UI is a Zooming/Zoomable User Interface (ZUI), which was inspired by Jef Raskin's suggestion that ZUIs are the next logical step in standard UI research. The desktop in A2 can be zoomed in or out to a significant degree, respectively enlarging or shrinking UI elements on the screen. This gives an overview of currently open windows, and offers a way to quickly pan the desktop.

4.4 Research Window Managers

Much like experimental operating systems, window management schemes arising from academic research shed practicality in exchange for attaining loftier aims.

Like many tiling window managers, maximizing the utilization efficiency of a user's screen and the amount of information they can fit on their screen is a frequent goal found in window management research. The Siemens' RTL window manager [117] was built upon the belief that automatic tiling would best optimize a user's screen space according to the number of windows available. By making estimated guesses about which windows were important and could be reduced in size to accommodate windows estimated to be more important, all windows could be fit on the screen while keeping most of the user's desired information on the screen. A two-dimensional space-tracking algorithm, called corner stitching [118], was used to make estimates based on how much screen space was empty and how much was used in addition to the relative importance of the windows. Other systems like Elastic Windows [119] and Bell and Feiner's work in *Dynamic Space Management for User Interfaces* [120] have also focused on algorithmically maximizing screen usage for user interfaces including window managers.

WinCuts [8] was also developed with screen utilization in-mind, but took an alternate approach, operating on the graphics of the windows as opposed to strictly the windows themselves. Rather than resizing the windows themselves to fit all information on the screen, the relevant parts of windows could be selected and mirrored in their own windows, WinCuts, for organization by the user. This provided the user with greater flexibility, as by reducing the amount of information on the screen to just what is needed, more information can be fit on the screen. Furthermore, WinCuts could be shared with other users and could be used to construct new interfaces based on the parts of the component interfaces in the WinCuts. Metisse also made use of directly manipulating the graphics of windows to present new types of interfaces and interactions within the desktop metaphor [121]. By allowing the user to perform typical graphical operations on windows, such as scaling or rotation, it opened up the possibilities for interactions such as zooming out a particular window, or rotating a window for viewing at a different perspective relative to the computer monitor.

5 Thesis Window Manager

The culmination of this thesis is the creation of a new window manager that implements the concepts found in state-of-the-art window managers and combines them on a platform that will allow for extensive testing with a diverse set of workflows. The resulting window manager is forked from StumpWM, and uses the Compton compositing manager [122] to perform all zooming functions.

5.1 Framework and Methodology

StumpWM was chosen as the base for the thesis window manager due to the flexibility afforded to it by Common Lisp. This proved helpful for the power of the language itself as well as its environment, where on-demand reloading made it possible to test new functionality without reloading the window manager. StumpWM's flexibility is also important to the goal of the thesis window manager to allow for flexible management of windows.

The thesis window manager presents a ZUI interface similar to A2: the desktop can be freely zoomed, panned, and manipulated directly, without need for any special interfaces for interaction such as a pager. This functionality manifests itself in two modes, which are largely invisible to the user. When no scaling is applied to the desktop, all windows can be interacted with normally. Clicking on the desktop and dragging along it moves all currently mapped windows, effectively panning the desktop. However, upon using the scroll-wheel on the desktop, the "overview" mode is engaged wherein scaling is applied to all windows in the current workspace. This causes the graphics for each currently-displayed image to shrink, and therefore allows for more windows to be displayed on the physical screen. For simplicity, windows cannot be interacted with using the mouse in overview mode. However, they can be moved and resized anywhere in the currently visible area, and the desktop can be scaled as before. To re-enter the normal mode, a user simply has to set the desktop to be unscaled. As previously mentioned, the two modes are largely invisible to the user in that the transition between them only changes how the user interacts with the windows, and does not produce any visual effects that would distinctively separate them; the user feels as if they are directly interacting with the desktop at all times.

A compositor is needed to manipulate window graphics in this way, which stores the image data for each window in a way that is accessible by other functions within the display server or an external program for manipulation. In X11, this is accomplished through the Composite extension, which allows the programmer to request for a window's pixmap, the X11 native image data format, to be made available in memory for manipulation by the compositor. To accomplish this, Compton was used as an external compositor to scale the window pixmaps and draw the resulting image to the screen. Compton was chosen since it is an independent program and fairly small, measuring at around 10,000 lines of C code. To communicate when Compton should scale, its DBUS interface was extended to include functions that specify the desktop scale, which is utilized by StumpWM when a user chooses a new scale.

To leverage the diverse functions offered by the thesis window manager with a mouse, an editable text field, the command bar, is added to each window as a window decoration. It was repurposed from the StumpWM input field, and offers most of the same keyboard interactions and rendering capabilities, but holds text in its buffer even after execution and is operable through the mouse. The command bar's interactions are similar to the tag pane in acme and universal text commands in Oberon, with a left click allowing the user to select text and place the cursor, and a middle click selecting the current word, delimited by spaces. Text executed in the command bar is executed entirely at runtime, and can consist of pre-defined commands, StumpWM commands, Lisp functions, or terminal commands. Commands are specified in a hash table and simply map arbitrary text to a specified command with a preferred method of evaluation, i.e. the Lisp function that will evaluate the command.

The command bar is typically placed on the top of the window through reparenting the bar and the window to a single parent window, making it functionally identical to bars in most

floating window managers. However, it functions as an independent window, and can also be placed on the top of the screen as a dock. All command bars have their own text buffer and can correspondingly be edited independently from other bars. The initial text in a command bar can be set through a global variable configurable by the user. Following English-language conventions, text is left-justified, making the command bar resemble the window decorations of window managers like macOS, Unity, and TWM. The command bar spans the window's whole width, and text can be entered over the entire width. As an anchor for moving and resizing the window, a character has been placed on the leftmost location on the window, usable through the mouse.

5.2 Algorithms and Features

The fundamental model used to organize windows in the thesis window manager is each window's dimensions, labeled as its geometry, which consists of the window's Cartesian coordinates at its upper-left corner, and the coordinates at its lower-right corner. These are organized into lists for when all windows must be processed and a graph structure when a more directed approach is desired. An infinite space in each axis is assumed, where the actual screen's dimensions are not considered.

To make it easy to view all windows in a particular set within the bounds of the screen, a trivial overview algorithm was written that returns the coordinates of the rectangle containing all windows in the set. In essence, the coordinates of the furthest points in the negative x and positive y axes and in the positive x and negative y axes are returned. This algorithm is sufficiently optimal for its use, running in $O(n)$ for an input of n windows.

These overviews are tied directly into views, a feature similar to the doors used in

VTWM. A view consists of a reference window, that window's current coordinates, and a specified scale. To activate a view, the window's new coordinates and its previous coordinates are subtracted to obtain a coordinate delta between the current position in the desktop and the coordinates of the view. The scale is then simply set to finish activating the view.

Simple tiling and maximizing features are offered for automatically sizing windows in relation to the screen size. Windows can be resized to any divisor of the screen's space, and can be anchored to any side of the screen. This scheme allows for basic manual static tiling layouts, including full-screen, vertical, horizontal, and grid tiling.

A key component of the thesis window manager is the automatic placement of new windows, which allows for easier and faster navigation when creating a new window. Ordinarily, a new window will spawn within the current screen, which in many cases will be above the currently-viewed window. This then requires navigating the desktop to find a suitable place for the window and move it. To relieve the user of this burden and optimize usage of the local desktop area, windows are automatically placed in an open space found by searching around the root of the current window's cluster.

Each cluster consists of a graph of windows centered around a root node. Each node has four edges: one for each side of the window, which specifies the window closest to that window and its distance. If there is no window within the current cluster on that side of the window, it is labeled as being an infinite distance.

To place the new window and construct the graph, a breadth-first search is performed on the graph of the current cluster, where windows are added in a counter-clockwise fashion starting at the root window. Each edge coming out of each window is tested to see whether it represents a space large enough to hold the new window. If it is, a temporary geometry is placed within the

edge and is then tested for overlap against all other windows in the cluster. If this is successful, the geometry is added as a candidate; otherwise, it is discarded. If the window at the other end of the edge has not been examined, any untested edges are then added to a queue in the aforementioned order. This process repeats until all edges have been tested.

6 Discussion

While the thesis window manager pulls substantially from previous research in the basics of its framework and ideas, it also contains novel ideas that may be used for window management or further research on window management.

The thesis window manager is task-centric, much like contemporary research window management systems, in that it focuses on individual tasks that may be composed of multiple programs as opposed to details around the specific programs that are run. It is intended that a user will have windows that comprise a single project or task close to one another in the virtual desktop. The effect of this is that the virtual desktop will consist of clusters laid across it, essentially composing spatially-arranged workspaces.

Another effect of this, and a guiding principle in the thesis window manager is its intended use for a substantial number of windows; as a guideline, more than twenty. The core feature that allows for this is the large desktop, which allows the user to place windows in a memorable way. In floating window managers, windows are simply placed over each other, which leaves them visually indistinct when an open window is selected again. This is aided by the ZUI, which offers a way to easily view and manipulate these window positions while retaining the spatial relationship between the windows. The need for this is clear: the amount of RAM available in

consumer computers has increased substantially since the advent of the GUI, but despite this no window managers have focused on managing an arbitrarily large number of windows.

This spatial relationship between windows also affords this model another useful property: the encoding of Cartesian coordinates within the windows. The algorithms used for automatically organizing windows rely on this as a means to derive information about the windows. Furthermore, it opens up the possibility for framing a number of window management problems as standard machine learning problems. Since the relationships between the windows have a clear visual aspect for the user, as well as a data-centric aspect for any algorithms, this property is helpful in translating a user's intentions into data for an algorithm.

Despite the power afforded to users by most tiling window managers in comparison to floating window managers, many completely lack discoverability. That is to say that a user using a tiling window manager without any prior experience will have an exceptionally difficult time using it without reading the manual, as there are no on-screen cues to indicate how to operate it. Even long-term users may find themselves having difficulties if keyboard shortcuts are forgotten. While keyboard shortcuts afford considerable gains in speed due to the time involved in aiming and moving a cursor as opposed to pressing a key combination, the thesis window manager uses a more mouse-centric model to make features accessible without prior knowledge. Furthermore, the text-based nature of window decoration buttons makes their function clear; if necessary, these can also be remapped to labels that are more intuitive for a particular user.

Many modern tiling window managers aim to automatically perform some management capabilities for the user in an effort to reduce friction when using the window manager. The thesis window manager has extended this to placing and classifying windows, by letting the user place a window without regard for placement in a grid system, and by classifying a window for the user.

Ordinarily, windows must either be placed within a statically-sized frame or require specification of a workspace or tag. In the thesis window manager, placement within a grid, as in systems like VTWM or FVWM, is not necessary, allowing less focus on where a window is placed and how it aligns. Furthermore, determining workspace a window belongs in is often unnecessary, as the window placement algorithm will simply place it in the current cluster.

While the thesis window manager improves on other state-of-the-art window managers in its goals, it also necessitated trade-offs to better improve those goals. At a core level, it was developed for Linux (though should be compatible with other Unix-like operating systems), as opposed to a research operating system like A2 or Plan 9. The reasons for this ultimately relate to the popularity behind Linux.

From the perspective of a developer, X (especially on Linux) is a well-documented system with numerous tools and libraries, owing to its availability for over thirty years and large mindshare. This makes it somewhat easier to develop for, and offers a significant amount of code for learning and re-use as a nature of the open development model most programmers use. These advantages are in contrast to systems like Plan 9, which despite their architectural elegance have largely remained a strong interest for a small core of devoted developers, and a curiosity for a slightly larger group of others. The thesis window manager also did not test more sophisticated system-level changes, and so did not need the cleaner designs offered by research systems.

The X window system is also a strong choice from the user perspective. As the default display system on Unix-like operating systems, virtually all graphical programs for these systems have been written for it. Owing to the 2.33% desktop marketshare Linux possesses, Linux has shown itself to be a viable system for daily use by a significant number of users, which makes it a ripe option for testing actual workflows outside of a specialized environment.

Despite these advantages, X is showing its age, and is planned for deprecation in favor of Wayland, a new display server and corresponding protocol designed to fix its architectural flaws. However, despite the benefits of developing on what would seem to be a more forward-thinking and future-proof system, Wayland has distinct disadvantages. The largest of which is the amount of code required to develop a Wayland compositor, the basic rendering mechanisms under the window manager. As TinyWM has shown, a simple window manager can be written in under 100 SLOC. However, the boilerplate code required for a Wayland compositor is much higher, and would not have added to testing window management techniques. Similarly, its immaturity means it is not yet the default system on many Linux distributions, and the tools available for it are fewer in number than their X11 counterparts.

7 Future Work

The thesis window manager only provides a framework and a basic set of functions to showcase this framework and provide a usable window manager. Further research would focus on features within this window manager that make use of the ideas presented in this thesis.

To separate different tasks, users may wish to group their windows into clusters of windows pertaining to a particular task, in a fashion similar to virtual desktops. To automatically determine which task a window belongs to, a clustering algorithm could be written to cluster “connected” windows, defined as being within a certain absolute distance of each other. In essence, a cluster is merely an index for a particular window graph, and does not hold any independent information on the windows it holds.

To construct a cluster, a list of clusters and a hash table mapping each individual window

back to its list index are initialized. The list of all windows currently mapped to the display is then processed to determine the initial clusters by its node, with edges in each direction to the closest window in that direction. If a window is connected to the current window, and has not been previously checked, it is added to the current window's cluster. Otherwise, it is placed in its own cluster. If the window has been checked, but is connected, it is moved to the cluster of the current window. This process repeats for each unchecked window until all windows have been checked, and the resulting list of clusters is returned.

To make navigating between windows more even as a product of better alignment, a compaction algorithm could be developed that shortens the spaces between windows. To accomplish this, it simply performs a breadth-first traversal of all nodes in the window graph, starting at the root node, and shortens any edges greater than a certain length between windows, updating the windows and their corresponding nodes on each compaction. The resulting window cluster would have a uniform amount of space between each window.

To fully make use of the compositing used in the thesis window manager, "portal" windows, as outlined by Ben Bederson in his work on Pad++ [123, 124], would be a very powerful feature to better integrate the ZUI with the desktop. Portal windows would entail a window containing an image rendered by the compositor that marks a certain scale and location in the desktop. Implementation could simply consist of rendering any window pixmaps contained in the region of the portal inside its window. Interactivity would mostly function as in the overview mode, and allow the user basic pager functionality.

Anchoring the position of windows relative to other items on the desktop would allow for more sophisticated relationships between windows at a user-determined level. These could include "sticky" windows that are not panned, and hold their size and location on the user's

monitor when the user pans the desktop. This could also possibly apply to scaling to control how windows are scaled as the user scales the desktop; however, this feature would need to be implemented with care due to the input issues involved with scaling. Windows could also be attached to other windows, in a fashion similar to Stack & Tile.

The algorithms used to process windows in the thesis window manager provide a good basic set of features, but also a path forward for future research. An interface by which the current state of the desktop acted as input for a function for manipulating the windows on a desktop would be useful in providing for new work. This would allow greater modularization in how windows are processed and provide greater flexibility in how these algorithms are applied. Interacting with an interface would provide a level of abstraction useful in prototyping these algorithms, and would allow users to swap them out as necessary. For example, different clustering algorithms could be used depending on desired behavior.

Research on additional algorithms to use in processing windows would be one of the more obvious, but more conceptually-involved additions to the framework outlined here. The explosion of machine learning research could likely be applied to this problem, as windows contain a significant amount of information about them aside from the coordinate-based algorithms that have been demonstrated. Reinforcement learning could potentially prove a useful method of learning which window a user would switch to next, by suggesting a window to switch to, but allowing for spontaneity in which window a user selects. Algorithms could also be devised which take a greater role in automatic organization of windows. An example algorithm would give the user different views of open windows on different virtual desktops. Views could entail clusters sorted by program or program-type, clusters arranged by how much different windows have been used, or could organize windows such that more frequently used programs gravitate toward the

center of each cluster.

The architecture of the system also could be a rewarding area for further research. The primary components of the thesis window manager are largely orthogonal in functionality, and could possibly be split into separate tools for window management, as in the case of WMutils. In this vein, the command bar provides a good basis for text commands and customization, but does not fully reach the power attained by Oberon or Acme. In Oberon, any available text item can be used as a command, as opposed to the limitation of the thesis window manager to only text in a command bar. This could likely be accomplished with an X11 function call that captures any selected text. In Acme, the core philosophy behind text commands is the ability to integrate with the existing system. Text commands in Acme take their input and return their output to it, allowing external programs to be called within Acme that are also usable as general-purpose tools. However, as the interface of the thesis window manager is graphically-based and not text-based, the majority of currently-available command-line tools are not particularly useful. Despite this, the flexibility of the X11 protocol could allow for the thesis window manager to pipe tools information about the current desktop, and have their output applied to the desktop. Output could also be pushed to an external window or even to the tag bar itself, for example as in a function which reads from the system clock and outputs it to its command bar. These functions would allow the thesis window manager to, like Acme, integrate with the existing system and make the command bar a more powerful component.

A dock would also be highly useful to the thesis window manager as an interface component to provide more mouse-based functionality. This could include features that are not currently present in X11-based docks by taking advantage of large desktop functionality. The most obvious of these is a tree-style window list that indents or colors windows based on their

cluster, giving the user a quick overview of all open windows that could likely be scanned more quickly than images, and could potentially also offer basic interaction functionality, such as closing or moving windows.

8 Conclusions

This thesis has presented an academic survey of the architectures and designs of X11 Linux window managers, both popular and lesser-known, in addition to other display systems on Linux and other operating systems. It was found that there are very strong lines of heritage in X window managers, owing to many of them being free software with easily accessible codebases. Furthermore, the simplicity of some of these window managers has given way to numerous forks, some of which improve upon the concepts implemented in their predecessor. All X window managers fall under the categories of floating or tiled, though most tiled window managers support floating windows, and furthermore that tiling comes in dynamic and manual flavors, with numerous dynamic algorithms by which to arrange windows. Some window managers also accept scripts from users, providing a layer of abstraction that allows for a high level of customization.

In the future, window manager developers will likely move to developing for Wayland, which has been designed for use on personal computers, as opposed to the terminal-mainframe style computing that guided the design of X. The structure of Wayland will require complete rewrites of current window managers, but provides new opportunities through its ability to natively composite windows.

The window manager presented by this thesis has taken into account ideas from both academic and non-academic sources to provide features not available in any single window

manager. Of note are its combination of an Oberon-style textual command bar with a pannable large desktop and compositing to provide easy viewing and navigation of this desktop. To take advantage of these features, algorithms were developed which operate on windows across a large virtual desktop. Its compatibility with X, the default display system on Unix-like operating systems, and intention for public availability under the GNU General Public license will encourage further research. Features regarding the algorithms used to process windows and the structure of the window manager have been proposed as potential places for further research.

References

[1] Vannevar Bush. *As We May Think*. https://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/. 1945.

[2] Douglas Englebart. *Augmenting Human Intellect: A Conceptual Framework*. https://www.dougengelbart.org/pubs/augment-3906.html. 1962.

[3] Douglas Englebart. *The Demo: 1968*. http://www.dougengelbart.org/firsts/dougs-1968-demo.html. 1968.

[4] Xerox Palo Alto Research Center. *Alto User's Handbook*. 3333 Coyote Hill Road, Palo Alto, CA, 94304, 1979.

[5] Inc. Apple. *Mac OS X Leopard - Features - Spaces*. https://web.archive.org/web/20070710195557/http://www.apple.com/macosx/leopard/features/spaces.html. 2007.

[6] Richie Fang. *Virtual Desktops in Windows 10 - The Power of Windows...Multiplied*. https://blogs.windows.com/windowsexperience/2015/04/16/virtual-desktops-in-windows-10-the-power-of-windowsmultiplied/. 2015.

[7] Michael S Bernstein, Jeff Shrager, and Terry Winograd. "Taskposé: exploring fluid boundaries in an associative window visualization". In: *Proceedings of the 21st annual ACM symposium on User interface software and technology*. ACM. 2008, pp. 231–234.

[8] Desney S Tan, Brian Meyers, and Mary Czerwinski. "WinCuts: manipulating arbitrary window regions for more effective use of screen space". In: *CHI'04 extended abstracts on Human factors in computing systems*. ACM. 2004, pp. 1525–1528.

[9] Joona Antero Laukkanen. "A scalable and tiling multi-monitor aware window manager". In: *CHI'11 Extended Abstracts on Human Factors in Computing Systems*. ACM. 2011, pp. 911–916.

[10] Thomas M. Frey. "Bluebottle: A Thread-safe Multimedia and GUI Framework for Active Oberon". en. PhD thesis. ETH Zürich, 2005. DOI: 10.3929/ethz-a-004999723.

[11] ETH Zürich. *Oberon Community Platform*. http://www.ocp.inf.ethz.ch/wiki/Documentation/Oberon. 2007.

[12] Clemens Zeidler, Christof Lutteroth, and Gerald Weber. "An evaluation of stacking and tiling features within the traditional desktop metaphor". In: *IFIP Conference on Human-Computer Interaction*. Springer. 2013, pp. 702–719.

[13] Rob Pike. http://doc.cat-v.org/plan_9/3rd_edition/rio/.

[14] David Hogan. *9wm*. http://unauthorised.org/dhog/9wm.html. 1996.

[15] *8 1/2*. http://doc.cat-v.org/plan_9/4th_edition/papers/812/.

[16] Tuomo Valkonen. *A tiling tabbed window manager designed for keyboard users. This repository is intended to make the last official ion3 release available because Tuomo has abandoned it.* https://github.com/jan0sch/Ion3. 2009.

[17] Jef Raskin. *The humane interface: new directions for designing interactive systems*. Addison-Wesley Professional, 2000.

[18] *Window manager*. https://wiki.archlinux.org/index.php/window_manager. 2018.

[19] *Comparison of tiling window managers*. https://wiki.archlinux.org/index.php/Comparison_of_tiling_window_managers.

[20] *Comparison of X window managers*. https://en.wikipedia.org/wiki/Comparison_of_X_window_managers. 2017.

[21] *Guide to X11/Window Managers*. https://en.wikibooks.org/wiki/Guide_to_X11/Window_Managers. 2017.

[22] Giles Orr. *The Comprehensive List of Window Managers for Unix*. http://gilesorr.com/wm/table.html. 2017.

[23] Giles Orr. *The Other Window Managers*. http://gilesorr.com/papers/otherwm2003/book1.html. 2003.

[24] Christopher Tronche. *The Xlib Manual*. https://tronche.com/gui/x/xlib/. 2005.

[25] among others Jasper St. Pierre. *Explanations - Play, don't show*. https://magcius.github.io/xplain/article/. 2017.

[26] *Index of /afs/athena/project/windowmanagers/dev/uwm*.
https://stuff.mit.edu/afs/athena/project/windowmanagers/dev/uwm/. 1997.

[27] Brian Proffitt. *From the Desktop: Tom LaStrange Speaks!*
http://www.linuxplanet.com/linuxplanet/reports/3000/1. 2001.

[28] Tom LaStrange. "An Overview of twm (Tom's Window Manager)". In: *Xhibition [Last89] Nahaboo, Colas, "GWM, The Generic X11 Window Manager," Xhibition* 89 (1989).

[29] X Consortium. *TWM Source Code*.
http://xwinman.org/archive/twm/twm-X11R6-xc-fix13.tar.gz. 1994.

[30] Matthew Fuller. *CTWM*. http://www.ctwm.org/index.html. 2017.

[31] Håkon Wium Lie. *Appendix A*.
https://www.w3.org/People/howcome/TEB/www/hwl_th_18.html. 1995.

[32] Callum Gibson and Seth Robertson. *vtwm*. http://www.vtwm.org/. 2013.

[33] https://github.com/da4089/tvtwm.

[34] Thomas E LaStrange. "swm: An X Window Manager Shell." In: *USENIX Summer*. 1990, pp. 299–306.

[35] Callum Gibson and Seth Robertson. *vtwm-5.4.7.tar.gz*.
http://sourceforge.net/projects/vtwm/files/vtwm-5.4.7.tar.gz. 2005.

[36] *Window Maker Home*. http://windowmaker.org. 2017.

[37] The GNUStep Project. *WindowMaker*. http://repo.or.cz/wmaker-crm.git.

[38] The GNUStep Project. *WindowMaker Tour*.
http://windowmaker.org/guidedtour/index.html.

[39] *Blackbox Wiki*. http://blackboxwm.sourceforge.net. 2006.

[40] *fluxbox.org*. http://fluxbox.org/. 2018.

[41] *Openbox*. http://openbox.org/wiki/Main_Page. 2018.

[42] http://man.cat-v.org/plan_9/1/rio.

[43] http://man.cat-v.org/plan_9/4/rio.

[44] David Hogan. *9wm*. https://github.com/9wm/9wm. 1996.

[45] Decklin Foster. *aewm*. http://www.red-bean.com/decklin/aewm/. 2007.

[46] Lars Bernhardsson. *Larswm*. http://porneia.free.fr/larswm/larswm.html. 2004.

[47] James Carter. *lwm*. http://www.jfc.org.uk/software/lwm.html.

[48] Benjamin Drieu. *w9wm*. http://www.drieu.org/code/w9wm.en.html. 2000.

[49] Nick Gravgaard. *Windowlab*. https://github.com/nickgravgaard/windowlab. 2016.

[50] Decklin Foster. *aewm source code*. http://www.red-bean.com/decklin/aewm/aewm-1.3.12.tar.bz2. 2007.

[51] Ciaran Anscomb. *evilwm*. http://www.6809.org.uk/evilwm/. 2015.

[52] Michael Cardell Widerkrantz. *mcwm*. http://hack.org/mc/hacks/mcwm/. 2015.

[53] venam. *A fast floating WM written over the XCB library and derived from mcwm*. https://github.com/venam/2bwm. 2017.

[54] Marius Aamodt Eriksen. *cwm*. https://github.com/chneukirchen/cwm. 2017.

[55] OpenBSD. *FAQ - The X Window System*. https://www.openbsd.org/faq/faq11.html#Intro. 2017.

[56] Marius Aamodt Eriksen. *CWM(1)*. https://man.openbsd.org/cwm. 2017.

[57] *wmi-11*. https://dl.suckless.org/wmi/wmi-11.tar.gz.

[58] Yaroslav Rastrigin. *TrsWM*. https://web.archive.org/web/20070104075513/http://yarick.territory.ru/trswm/. 2004.

[59] Jérémie Courrèges-Anglas. *Ratpoison*. http://www.nongnu.org/ratpoison/. 2017.

[60] https://web.archive.org/web/20120919235334/http://wmi.suckless.org/. 2012.

[61] Anselm R. Garbe. *wmii*. https://github.com/0intro/wmii.

[62] Suckless. *dwm - dynamic window manager*. https://git.suckless.org/dwm/. 2017.

[63] Suckless. *dwm - dynamic window manager*. https://dwm.suckless.org. 2017.

[64] *patches*. https://dwm.suckless.org/patches/.

[65] Rinaldini Julien (pyknite). *catwm is a very simple tiling window manager*. https://github.com/pyknite/catwm. 2011.

[66] moetunes. *A minimal dynamic tiling window manager built from catwm*. https://github.com/moetunes/dminiwm. 2014.

[67] Ivan Kanakarakis (c00kiemon5ter). *tiny but monstrous tiling window manager*. https://github.com/c00kiemon5ter/monsterwm. 2012.

[68] Jari Vetoniemi (Cloudef). *Port of monsterwm to xcb*. https://github.com/Cloudef/monsterwm-xcb. 2016.

[69] Robin Schroer (sulami). *Fast dynamic tiling window manager*. https://github.com/sulami/FrankenWM. 2017.

[70] Julien Danjou. *Announcing "awesome"*. http://article.gmane.org/gmane.comp.window-managers.dwm/3285/match=awesome. 2007.

[71] Uli Schlachter. https://github.com/awesomeWM/awesome.

[72] Sander Van Dijk Anslem R. Garbe. *2wm 0.1*. https://dl.suckless.org/misc/2wm-0.1.tar.gz. 2007.

[73] Michael Stapelberg. *i3wm*. https://i3wm.org.

[74] *Official Sawfish website*. http://sawfish.wikia.com/wiki/Main_Page. 2017.

[75] *Sawfish Window-Manager*. https://github.com/SawfishWM/sawfish. 2017.

[76] *Viewports*. https://sawfish.tuxfamily.org/sawfish.html/Viewports.html.

[77] *Qtile - A hackable tiling window manager written in Python*. http://www.qtile.org/. 2017.

[78] Elmo Mäntynen. *PYWM - your Python Window Manager*. http://pywm.sourceforge.net/. 2006.

[79] Elmo Mäntynen. *PyWM Source Code*. https://sourceforge.net/projects/pywm/files/pywm/0.1-1-a4-1/pywm-0.1-1-a4-1.tar.bz2/download. 2006.

[80] *A small, flexible, scriptable tiling window manager written in Python*. https://github.com/qtile/qtile. 2017.

[81] *Everything you need to know about Qtile*. http://docs.qtile.org/en/latest/. 2016.

[82] *xmonad | the tiling window manager that rocks*. http://xmonad.org.

[83] Spencer Janssen. *xmonad*. https://github.com/xmonad/xmonad. 2017.

[84] Wouter Swierstra. "Xmonad in Coq (Experience Report): Programming a Window Manager in a Proof Assistant". In: *SIGPLAN Not.* 47.12 (Sept. 2012), pp. 131–136. ISSN: 0362-1340. DOI: 10.1145/2430532.2364523. URL: http://doi.acm.org/10.1145/2430532.2364523.

[85] David Bjergaard. *stumpwm.github.io/*. https://stumpwm.github.io/. 2017.

[86] *The Stump Window Manager*. https://stumpwm.github.io/. 2017.

[87] *Extension Modules for StumpWM*. https://github.com/stumpwm/stumpwm-contrib. 2017.

[88] Nick Welch. *TinyWM*. http://incise.org/tinywm.html. 2005.

[89] Philippe Brochard. *CLFSWM*. https://common-lisp.net/project/clfswm/. 2012.

[90] Philippe Brochard. *A(nother) Common Lisp Full Screen Window Manager*. https://github.com/spacefrogg/clfswm. 2015.

[91] Michael Stapelberg. *i3*. https://github.com/i3/i3. 2017.

[92] Michael Stapelberg. *i3 User's Guide*. https://i3wm.org/docs/userguide.html. 2013.

[93] Michael Stapelberg. *i3-msg(1)*. https://build.i3wm.org/docs/i3-msg.html. 2012.

[94] Thorsten Wißmann. *herbstluftwm*. https://www.herbstluftwm.org/. 2017.

[95] *A manual tiling window manager for X11*. https://github.com/herbstluftwm/herbstluftwm. 2017.

[96] Thorsten Wißmann. *herbstluftwm(1)*. http://herbstluftwm.org/herbstluftwm.html. 2014.

[97] Bastien Dejean. *Simple X hotkey daemon*. https://github.com/baskerville/sxhkd. 2017.

[98] Bastien Dejean. *bspwm*. https://github.com/baskerville/bspwm. 2017.

[99] Antti Korpi. *experimental panning X window manager, with an infinite desktop*. https://github.com/anko/basedwm. 2015.

[100] *LiveScript - a language which compiles to JavaScript*. http://livescript.net/. 2016.

[101] Antti Korpi. *transparent fullscreen click-through WebKit browser window, for making cool desktop HUDs*. https://github.com/anko/hudkit. 2017.

[102] Chris Feng. *Emacs X Window Manager*. https://github.com/ch11ng/exwm. 2018.

[103] Chris Feng. *EXWM User Guide*. https://github.com/ch11ng/exwm/wiki. 2017.

[104] Patrick Haller. *Use X11 without a window manager*. https://github.com/patrickhaller/no-wm. 2017.

[105] dcat. *wmutils*. https://github.com/wmutils. 2017.

[106] *Set of window manipulation tools*. https://github.com/wmutils/core. 2017.

[107] *Useful bits and pieces*. https://github.com/wmutils/contrib. 2017.

[108] *Official FVWM repository*. https://github.com/fvwmorg/fvwm. 2017.

[109] *Operating System Market Share*. https://netmarketshare.com/operating-system-market-share.aspx?options=%7B%22filter%22%3A%7B%22%24and%22%3A%5B%7B%22deviceType%22%3A%7B%22%24in%22%3A%5B%22Desktop%2Flaptop%22%5D%7D%7D%5D%7D%2C%22dateLabel%22%3A%22Trend%22%2C%22attributes%22%3A%22share%22%2C%22group%22%3A%22platform%22%2C%22sort%22%3A%7B%22share%22%3A-1%7D%2C%22id%22%3A%22platformsDesktop%22%2C%22dateInterval%22%3A%22Monthly%22%2C%22dateStart%22%3A%222016-12%22%2C%22dateEnd%22%3A%222017-11%22%2C%22segments%22%3A%22-1000%22%7D. 2017.

[110] *Wayland*. https://wayland.freedesktop.org/.

[111] Drew DeVault et al. *Sway*. http://swaywm.org/.

[112] Drew DeVault. *i3 feature support*. https://github.com/swaywm/sway/issues/2. 2016.

[113] Preston Carpenter et al. *Way Cooler*. https://github.com/way-cooler/way-cooler.

[114] Preston Carpenter et al. *Customizable Wayland compositor (window manager)*. https://github.com/way-cooler/way-cooler. 2018.

[115] Drew DeVault et al. *i3-compatible Wayland compositor*. https://github.com/swaywm/sway. 2018.

[116] Drew DeVault. *The future of Wayland, and sway's role in it*. http://sircmpwn.github.io/2017/10/09/Future-of-sway.html. 2017.

[117] Ellis S Cohen et al. “Automatic strategies in the Siemens RTL tiled window manager”. In: *Computer Workstations, 1988., Proceedings of the 2nd IEEE Conference on*. IEEE. 1988, pp. 111–119.

[118] Dileep A Divekar and Richard I Dowell. “Corner stitching: A data-structuring technique for VLSI layout tools”. In: *IEEE Transactions on Computer-Aided Design* 3.1 (1984), p. 87.

[119] Eser Kandogan and Ben Shneiderman. “Elastic Windows: evaluation of multi-window operations”. In: *Proceedings of the ACM SIGCHI Conference on Human factors in computing systems*. ACM. 1997, pp. 250–257.

[120] Blaine A. Bell and Steven K. Feiner. “Dynamic Space Management for User Interfaces”. In: *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*. UIST ’00. San Diego, California, USA: ACM, 2000, pp. 239–248. ISBN: 1581132123. DOI: 10.1145/354401.354790. URL: http://doi.acm.org/10.1145/354401.354790.

[121] Olivier Chapuis and Nicolas Roussel. “Metisse is not a 3D desktop!” In: *Proceedings of the 18th annual ACM symposium on User interface software and technology*. ACM. 2005, pp. 13–22.

[122] Christopher Jeffrey. *A compositor for X11*. https://github.com/chjj/compton. 2017.

[123] Ben B Bederson, Larry Stead, and James D Hollan. “Pad++: Advances in multiscale interfaces”. In: *Conference companion on Human factors in computing systems*. ACM. 1994, pp. 315–316.

[124] Ben Bederson and Jonathan Meyer. “Implementing a zooming user interface: experience building Pad++”. In: ().