The wmii User Guide

Kris Maglione

23 May 2009

Contents

1 Introduction

wmii is a simple but powerful window manager for the X Window System. It provides both the classic ("floating") and tiling ("managed") window management paradigms, which is to say, it does the job of managing your windows, so you don't have to. It also provides programability by means of a simple file-like interface, which allows the user to program in virtually any language he chooses. These basic features have become indispensible to the many users of wmii and other similar window managers, but they come at a cost. Though our penchant for simplicity makes wmii's learning curve significantly shorter than most of its competitors, there's still a lot to learn. The rest of this guide will be devoted to familiarizing new users with wmii's novel features and eccentricities, as well as provide advanced users with an in-depth look at our customization facilities.

1.1 Concepts

As noted, wmii provides two management styles:

Managed This is the primary style of window management in wmii. Windows managed in this style are automatically arranged by wmii into columns. Columns are created and destroyed on demand. Individual windows in the column may be moved or resized, and are often collapsed or hidden entirely. Ad-hoc stacks of collapsed and uncollapsed windows allow the user to efficiently manage their tasks. When switching from an active to a collapsed window, the active window collapses, and the collapsed one effectively takes its place.

Managed windows have an unadorned titlebar:

Floating Since some programs aren't designed in ways conducive to the managed work flow, wmii also provides the classic "floating" window management model. In this model, windows float above the managed windows, and may be moved freely about. Other than automatic placement of new windows and snapping of edges, wmii doesn't manage floating windows at all.

Floating windows are indicated by a decorated titlebar:

Fullscreen Fullscreen mode is actually a subset of the floating style. Windows may be toggled to and from fullscreen mode at will. When fullscreen, windows reside in the floating layer, above the managed windows. They have no borders or titlebars, and occupy the full area of the screen. Other than that, however, they're not special in any way. Other floating windows may appear above them, and the user can still select, open, and close other windows at will.

1.1.1 The Filesystem

All of wmii's customization is done via a virtual filesystem. Since the filesystem is implemented in the standardized 9P protocol, it can be accessed in many ways. wmii provides a simple command-line client, wmiir, but many alternatives exist, including libraries for Python, Perl, Ruby, PHP, and C. It can even be mounted, either by Linux's 9p.ko kernel module or indirectly via FUSE.

The filesystem that wmii provides is "virtual", which is to say that it doesn't reside on disk anywhere. In a sense, it's a figment of wmii's imagination. Files, when read, represent wmii's current configuration or state. When written, they perform actions, update the UI, etc. For instance, the directory /client/ contains a directory for each window that wmii is currently managing. Each of those directories, in turn, contains files describing the client's properties (its title, its views[1], its state). Most files can be written to update the state they describe. For instance, /client/sel/ctl describes the state of the selected client. If a client is fullscreen, it contains the line:

```
Fullscreen on
```

To change this, you'd update the file with the line `Fullscreen off` or even `Fullscreen toggle` to toggle the client's fullscreen state.

The concept of controlling a program via a filesystem derives from Plan 9, where such interfaces are extensive and well proven. The metaphor has shown itself to be quite intuitive to Unix users, once the shock of a "virtual" filesystem wears off. The flexibility of being able to control wmii from myriad programming languages, including the standard Unix shell and even from the command line, is well worth the shock.

1.1.2 Views and Tags

Like most X11 window managers, wmii provides virtual workspaces. Unlike other window managers, though, wmii's workspaces are created and destroyed on demand. Instead of being sent to a workspace, windows in wmii are tagged with any number of names. Views are created dynamically from these tags, and automatically if the user tries to access them. For instance, if a window is given the tags 'foo' and 'bar', the two views 'foo' and 'bar' are created, if they don't already exist. The window is now visible on both of them. Moreover, tags can be specified as regular expressions. So, a client tagged with /^foo/ will appear on any view named 'foo', 'foo:bar', and so forth. Any time a client is tagged with a matching tag, or the user opens a matching view, the window is automatically added to it.

1.1.3 The Bar

wmii provides a general purpose information bar at the top or bottom of the screen. The bar is divided into a left and a right section. Each section is made up of buttons, with a single button spanning the gap between the two sides. Buttons can be individually styled, and can hold any text content the user wishes. By convention, the buttons to the left show view names, and those to the right display status information.

[1] Views in wmii are akin to workspaces or virtual desktops in other window managers, but with some subtle differences.

1.1.4 The Menus

wmii includes two simple, external menu programs. The first, wimenu, is keyboard-based, and is used to launch programs and generally prompt the user for input. It provides a list of completions which are automatically filtered as you type. The second, wi9menu, is mouse-based, and is generally used to provide context menus for titlebars and view buttons. Both menus can be easily launched from shell scripts or the command line, as well as from more complex scripting languages.

1.1.5 The Keyboard

wmii is a very keyboard friendly window manager. Most actions can be performed without touching the mouse, including launching, closing, moving, resizing, and selecting programs. New keybindings of any complexity can easily be added to handle any missing functionality, or to simplify any repetitive tasks.

1.1.6 The Mouse

Despite being highly keyboard-accessible, wmii strives to be highly mouse accessible as well. Windows can be moved or resized by dragging their window borders. When combined with a key press, they can be moved, resized, or raised by dragging any visible portion of the window. Mouse menus are accessed with a single click and drag. View buttons in the bar and client titlebars respond to the mouse wheel; view buttons can be activated by dragging any draggable object (e.g., a file from a file manager) over them.

2 Getting Started

This section will walk you through your first `wmii` startup. For your first experience, we recommend running `wmii` in its own X session, so you can easily switch back to a more comfortable environment if you get lost. Though you may start `wmii` from a session manager in your day to day use, these instructions will use `xinit`. To begin with, copy this file to your home directory, so we can open it in your new X session. Then setup your `~/.xinitrc` as follows:

```
cd

# Start a PDF viewer with this guide. Use any viewer
# you're comfortable with.
xpdf wmii.pdf &

# Launch wmii
exec wmii

# That was easy.
```

Before you run `xinit`, make sure you know how to switch between terminals. Depending on your system, your current X session is probably on terminal 5 or 7. You should be able to switch between your terminals by pressing Ctrl-Alt-F⟨*n*⟩. Assuming that your current X session is on terminal 7, you should be able to switch between it and your new session by pressing Ctrl-Alt-F7 and Ctrl-Alt-F8. Now you should be ready to start `wmii`. When you run the following command, from a terminal, you should be presented with a new X session, running wmii, with this document open in a PDF viewer occupying most of the screen. When you're there, come back to this page and continue. Now, open a terminal and execute:

```
xinit
```

2.1 Your First Steps

If everything went according to plan, you should be viewing this from a nearly empty `wmii` session. We're going to be using the keyboard a lot, so let's start with a convention for key notation. We'll be using the key modifiers Control, Alt, Shift, and Meta[1], which we'll specify as C-, A-, S-, and M-, respectively. So, ⟨C-S-a⟩ means pressing 'a' while holding `Control` and `Shift`. We'll also express mouse clicks this way, with ⟨M-Mouse1⟩ signifying a press of the right mouse button, with the Meta key depressed. Buttons 4 and 5 are the up and down scroll wheel directions, respectively.

[1]The Windows® key on most keyboards. The Penguin key, on the more tongue in cheek varieties.

2.1.1 Floating Mode

Begining with what's familiar to most users, we'll first explore floating mode. First, we need to select the floating layer. Press ⟨M-Space⟩. You should see the titlebar of this window change color. Now, press ⟨M-Return⟩ to launch a terminal. The easiest way to drag the terminal around is to press and hold ⟨M-Mouse1⟩ over the window and simply drag the window around. You should be able to drag the window anywhere onscreen without ever releasing the mouse button. As you drag near the screen edges, you should notice a snap. If you try to drag the window fully off-screen, you'll find it constrained so that a portion always remains visible. Now, release the window and move the mouse toward one of its corners. Press and hold ⟨M-Mouse3⟩[2]. As you drag the mouse around, you should see the window resized accordingly.

To move the window without the modifier key, move the pointer over the layout box to the left of its titlebar. You should see the cursor change. Now, simply click and drag. To resize it, move the pointer toward the window's edge until you see the cursor change, and again, click and drag. Now, to close the window, move the mouse over the windows titlebar, press and hold ⟨Mouse3⟩, select `Delete`, and release it. You should see this window's titlebar return to its original color, indicating that it's regained focus.

2.1.2 Managed Mode

Now, for the fun part. We'll start exploring managed mode by looking at the basics of columns. In the default configuration, columns have three modes:

Stack ⟨M-s⟩ The default mode for new columns. Only one window is fully visible per column at once. The others only display their title bars. When new windows are added to the column, the active window collapses, and the new one takes its place. Whenever a collapsed client is selected, the active window is collapsed to take its place.

Max ⟨M-m⟩ Like stack mode, but the titlebars of collapsed clients are hidden.

Default ⟨M-d⟩ Multiple uncollapsed windows may be visible at once. New windows split the space with the other uncollapsed windows in their vicinity. Windows may still be collapsed by shrinking them to the size of their titlebars. At this point, the behavior of a stack of collapsed and uncollapsed clients is similar to that of stack mode.

Before we open any new windows in managed mode, we need to explore the column modes a bit. Column modes are activated with the key bindings listed above. This column should be in stack mode now. Watch the right side of the titlebar as you press ⟨M-m⟩ to enter max mode. You should see an indicator appear. This tells you the number of hidden windows directly above and below the current window, and its position in that stack. Press ⟨M-d⟩ to enter default mode. Now we're ready to open another client. Press ⟨M-Return⟩ to launch another terminal. Now, press ⟨M-S-l⟩ to move the terminal to a new column to the right of this one. Once it's there, press ⟨M-Return⟩ two more times to launch two more terminals. Now that you have more than one window in a column, cycle through the three column modes again until they seem familiar.

[2]The right button.

2.1.3 Keyboard Navigation

To begin, switch back to default mode. The basic keyboard navigation keys, ⟨M-h⟩, ⟨M-j⟩, ⟨M-k⟩, and ⟨M-l⟩, derive from vi, and represent moving left, down, up, and right respectively. Try selecting each of the four windows currently visible on screen. Notice that navigation wraps from one side of the screen to the other, and from the top to the bottom. Now, return to the write column, switch to stack mode, and select each of the three terminals again. Do the same in max mode, paying careful attention to the indicator to the right of the titlebar.

Now that you can select windows, you'll want to move them around. To move a window, just add the Shift key to the direction keys. So, to move a window left, instead of ⟨M-h⟩, type ⟨M-S-h⟩. Now, experiment with moving windows, just as you did with navigating them, in each of the three column modes. Once you're comfortable with that, move a window to the floating layer. Since we toggled between the floating and managed layers with ⟨M-Space⟩, we'll move windows between them with ⟨M-S-Space⟩. Try moving some windows back and forth until it becomes familiar. Now, move several windows to the floating layer and try switching between them with the keyboard. You'll notice that ⟨M-h⟩ and ⟨M-l⟩ don't function in the floating layer. This is for both historical and logistical reasons. ⟨M-j⟩ and ⟨M-k⟩ cycle through floating windows in order of their most recent use.

2.1.4 Mouse Navigation

`wmii` uses the "sloppy focus" model, which is to say, it focuses windows when the mouse enters them and when you click them. It focuses windows only when you select them with the keyboard, click their titlebars, or press click them with ⟨M-Mouse2⟩. Collapsed windows may be opened with the mouse by clicking their titlebars. Moving and resizing floating windows should be largely familiar, and has already been covered. The same can't be said for managed windows.

Let's begin working with the mouse in the managed layer. Return to a layout with this document in a column on the left, and three terminals in a column to the right. Switch the right column to default mode. Now, bring the mouse to the top of the third terminal's titlebar until you see a resize cursor. Click and drag the titlebar to the very top of the screen. Now, move the cursor to the top of the second terminal's titlebar and drag it to the very bottom of the screen. Press ⟨M-d⟩ to restore the terminals to their original sizes. Now, click and hold the layout box of the second terminal. Drag it to the middle of the terminal's window and release. Click and hold the layout box of the third terminal and drag it to the middle of the first terminal's window. Finally, drag the first terminal's layout box to halfway down this window. ⟨M-Mouse1⟩ works to the same effect as dragging the layout box, but allows you to click anywhere in the window.

Now that you've seen the basics of moving and dragging windows, let's move on to columns. Click and drag the border between the two columns. If that's a difficult target to click, there's a triangle at the top of the division between the two columns that you can click and drag as well. If that's still too hard a target, try using ⟨M-Mouse3⟩, which works anywhere and provides much richer functionality.

2.1.5 Window Focus and Selection

For the purposes of keyboard navigation, `wmii` keeps track of which window is currently selected, and confers its titlebar a different color from the unselected windows. This window is the basis of relative motion commands, such as "select the window to the left", and the target of commands such as "close this window". Normally, the selected window is the same as the focused window, i.e., the window that recieves keyboard events. Some applications, however, present strange corner cases.

Focused, selected window This is the normal case of a window which is both selected and has the keyboard focus.

Unfocused, unselected window This is the normal case for an unselected window which does not have the keyboard focus.

Unfocused, selected window This is the first unusual case. This is the selected window, for the purposes of keyboard navigation, but it doesn't recieve keyboard events. A good example is an onscreen keyboard, which will recieve mouse clicks and translate them to keyboard events, but doesn't receive those keyboard events itself. Other examples include any window whilst another (such as `wimenu`) has grabbed the keyboard.

Focused, unselected window This is the second unusual focus case. The window has the keyboard focus, but for the purposes of keyboard navigation, it's not considered selected.

2.2 Running Programs

You've already seen the convenient key binding to launch a terminal, but what about other programs? To get a menu of all of the executables in your path, type ⟨M-p⟩. This should replace the bar at the bottom of the screen with a prompt, followed by a string of completions. Start typing the name of a program that you want to open. You can press ⟨Tab⟩ and ⟨S-Tab⟩ to cycle through the completions, or you can just press ⟨Return⟩ to select the first one. If you want to execute a more complex command, just type it out and press ⟨Return⟩. If you want to recall that command later, use `wimenu`'s history. Start typing the command you want and then press ⟨C-p⟩ until you come to it.

When you're done with a program, you'll probably want an easy way to close it. The first way is to ask the program to close itself. Since that can be tedious (and sometimes impossible), `wmii` provides other ways. As mentioned, you can right click the titlebar and select `Delete`. If you're at the keyboard, you can type ⟨M-S-c⟩. These two actions cause

wmii to ask nicely that the program exit. In those sticky cases where the program doesn't respond, wmii will wait 10 seconds before prompting you to kill the program. If you don't feel like waiting, you can select Kill from the window's titlebar menu, in which case wmii will forcefully and immediately kill it. Beware, killing clients is a last resort. In cases where the same program opens multiple windows, killing one will kill them all—without warning.

2.3 Using Views

As already noticed, wmii's concept of virtual workspaces is somewhat unique, so let's begin exploring it. Open up a terminal and press ⟨M-S-2⟩. You should see a new button on the bar at the bottom of the screen. When you click it, you should see your original terminal. Press ⟨M-1⟩ to come back here. Now, press ⟨M-3⟩, and ⟨M-1⟩ again to return here once more. Notice that the views were created when needed, and destroyed when no longer necessary. If you want to select a view with a proper name, use ⟨M-t⟩ and enter the name. Other than the dynamic creation of views, this is still similar to the familiar X11 workspace model. But that's just the begining of wmii's model. Open a new terminal, and type:

```
echo 'Hello world!'
```

Now, type ⟨M-S-t⟩. In the menu that appears, enter 1+2+3. Now, visit the views 1, 2, and 3, and you'll see the client on each. To remove a tag, type ⟨M-S-t⟩ again, and this time enter -2. You'll notice that the client is no longer on the 2 view. Finally, tag names needn't be discrete, ordinary strings. They can also be regular expressions. Select the terminal again, and enter +/^5/. Now, switch to the 5 view. Now try the 6 view. Finally, type ⟨M-t⟩ and enter 50 to check the 50 view. Clients tagged with regular expressions are attached to any matching views when they're created. So, when you switch to an empty view, or tag a client with a new tag, any clients with matching regular expressions are automatically added to it. When all explicitly tagged clients disappear from the view, and it's no longer visible, clients held there by regular expressions are automatically removed.

2.4 Learning More

For full tables of the standard key bindings, and descriptions of the precise semantics of the topics discussed above, you should refer to wmii's man pages.

3 Customizing wmii

There are several configuration schemes available for wmii. If you're only looking to add basic key bindings, status monitors, *et cetera*, you should have no trouble modifying the stock configuration for your language of choice. If you're looking for deeper knowledge of wmii's control interface, though, this section is for you. We'll proceed by building a configuration script in POSIX sh syntax, and move on to a discussion of the higher level constructs in the stock configuration scripts.

3.1 Events

The wmii control interface is largely event driven. Each event is represented by a single, plain-text line written to the /event file. You can think of this file as a named pipe. When reading it, you won't receive an EOF[1] until wmii exits. Moreover, any lines written to the file will be transmitted to all of its readers. Notable events include key presses, the creation and destruction of windows, and changes of focus and views.

We'll start building our configuration with an event processing framework:

```
⟨⟨Event Loop⟩⟩ ::=
# Broadcast a custom event
wmiir xwrite /event Start wmiirc

# Turn off globbing
set -f
# Open /event for reading
wmiir read /event |
# Read the events line by line
while read line; do
    # Split the line into words, store in $@
    set -- $line
    event=$1; shift
    line = "$(echo $line | sed 's/^[^ ]* //' | tr -d '\n')"
    # Process the event
    case $event in
    Start) # Quit when a new instance starts
        [ $1 = wmiirc ] && exit;;
    ⟨⟨Event Handlers⟩⟩
    esac
done
```

Now, we need to consider which types of events we'll need to handle:

[1] End of File

⟨⟨*Event Handlers*⟩⟩ ::=
⟨⟨*View Button Events*⟩⟩
⟨⟨*Urgency Events*⟩⟩
⟨⟨*Unresponsive Clients*⟩⟩
⟨⟨*Notice Events*⟩⟩
⟨⟨*Key Events*⟩⟩
⟨⟨*Client Menu Events*⟩⟩
⟨⟨*Tag Menu Events*⟩⟩

3.2 Bar Items

The bar is described by the files in the two directories `/lbar/` and `/rbar/` for buttons on the left and right side of the bar, respectively. The format of the files is:

⟨*Color Tuple*⟩ ⟨*Label*⟩

although the color tuple may be elided in cases where the label doesn't match its format. A ⟨*Color Tuple*⟩ is defined as:

⟨*tuple*⟩ ::= ⟨*foreground color*⟩ ⟨*background color*⟩ ⟨*border color*⟩
⟨*color*⟩ ::= #⟨*6 character RGB hex color code*⟩

Let's define our basic theme information now:

```
⟨⟨Theme Definitions⟩⟩ ::=
normcolors='#000000 #c1c48b #81654f'
focuscolors='#000000 #81654f #000000'
background='#333333'
font='drift,-*-fixed-*-*-*-*-9-*-*-*-*-*-*-*'
```

3.2.1 View Buttons

With a basic understanding of bar items in mind, we can write our view event handlers:

```
⟨⟨View Button Events⟩⟩ ::=
CreateTag)  # CreateTag ⟨Tag Name⟩
    echo $normcolors $1 | wmiir create /lbar/$1;;
DestroyTag) # DestroyTag ⟨Tag Name⟩
    wmiir rm /lbar/$1;;
FocusTag)   # FocusTag ⟨Tag Name⟩
    wmiir xwrite /lbar/$1 $focuscolors $1;;
UnfocusTag) # UnfocusTag ⟨Tag Name⟩
    wmiir xwrite /lbar/$* $normcolors $1;;
```

3.2.2 Urgency

Windows can specify that they require attention, and in X11 parlance, this is called urgency. When a window requests attention as such, or declares that it's been satisfied, `wmii` broadcasts an event for the client and an event for each view that it belongs to, and

fills in the client's layout box. It's the job of a script to decide how to handle it above and beyond that. The standard scripts simply mark urgent views with an asterisk:

```
⟨⟨Urgency Events⟩⟩ ::=
# The urgency events are 'Client' events when the program
# owning the window sets its urgency state. They're 'Manager'
# events when wmii or the wmii user sets the state.
UrgentTag)    # UrgentTag ⟨'Client' or 'Manager'⟩ ⟨Tag Name⟩
    wmiir xwrite /lbar/$2 *$2;;
NotUrgentTag) # NotUrgentTag ⟨'Client' or 'Manager'⟩ ⟨Tag Name⟩
    wmiir xwrite /lbar/$2 $2;;
```

3.2.3 Notices

The standard scripts provide a custom Notice event for displaying status information. The events appear in the long bar between the left and right sides for five seconds.

```
⟨⟨Notice Events⟩⟩ ::=
Notice)
    wmiir xwrite /rbar/!notice $line
    kill $xpid 2>/dev/null # Let's hope this isn't reused...
    { sleep 5; wmiir xwrite /rbar/!notice ' '; } &
    xpid = $!;;
```

3.3 Keys

Now to the part you've no doubt been waiting for:binding keys. When binding keys, you need to be aware of two files, `/keys` and `/event`. The former defines which keys `wmii` needs to grab, and the latter broadcasts the events when they're pressed. Key names are specified as a series of modifiers followed by a key name, all separated by hyphens. Valid modifier names are `Control`, `Shift`, `Mod1` (usually Alt), `Mod2`, `Mod3`, `Mod4` (usually the Windows® key), and `Mod5`. Modifier keys can be changed via `xmodmap(1)`, which is beyond the scope of this discussion. Key names can be detected by running `xev` from a terminal, pressing the desired key, and looking at the output (it's in the parentheses, after the keysym). A `wmii`-specific utility is forthcoming.

Examples of key bindings:

Windows® key + Capital A `Mod4-Shift-A`

Control + Alt + Space `Mod1-Control-Space`

Now, let's bind the keys we plan on using:

```
⟨⟨Bind Keys⟩⟩ ::=
{
cat <<!
Mod4-space
Mod4-d
```

```
Mod4-s
Mod4-m
Mod4-a
Mod4-p
Mod4-t
Mod4-Return
Mod4-Shift-space
Mod4-f
Mod4-Shift-c
Mod4-Shift-t
Mod4-h
Mod4-j
Mod4-k
Mod4-l
Mod4-Shift-h
Mod4-Shift-j
Mod4-Shift-k
Mod4-Shift-l
!
for i in 1 2 3 4 5 6 7 8 9 0; do
    echo Mod4-$i
    echo Mod4-Shift-$i
done
} | wmiir write /keys
```

and lay a framework for processing their events:

```
⟨⟨Key Events⟩⟩ ::=
Key) # Key ⟨Key Name⟩
    case $1 in
    ⟨⟨Motion Keys⟩⟩
    ⟨⟨Client Movement Keys⟩⟩
    ⟨⟨Column Mode Keys⟩⟩
    ⟨⟨Client Command Keys⟩⟩
    ⟨⟨Command Execution Keys⟩⟩
    ⟨⟨Tag Selection Keys⟩⟩
    ⟨⟨Tagging Keys⟩⟩
    esac;;
```

3.4 Click Menus

Sometimes, you have your hand on the mouse and don't want to reach for the keyboard. To help cope, wmii provides a mouse-driven, single-click menu. The default configuration uses it for client and tag menus.

```
⟨⟨Click Menu Initialization⟩⟩ ::=
clickmenu() {
    if res=$(wmii9menu -- "$@"); then eval "$res"; fi
}
```

3.5 Control Files

Most filesystem objects, including the root directory, have control files, named `ctl`. The first line of most control files is the canonical name of the directory they reside in, which comes in handy for the special `sel/` directories, which are aliases for the currently selected object of a group. The following lines represent properties of the object. Control files may be written to, in similar syntax to the values that can be read, to update those properties. For instance, if a file contains:

```
Fullscreen on
```

either of the following, when written to the file, will disable the `Fullscreen` state:

```
Fullscreen off
Fullscreen toggle
```

3.6 Clients

Clients are represented by directories under the `/client/` tree. Subdirectory names represent the client's X11 window ID. The special `sel/` directory represents the currently selected client. The files in these directories are:

ctl The control file. The properties are:

- **Fullscreen** The client's fullscreen state. When `on`, the client is displayed fullscreen on all of its views. Possible values are `on`, `off`, and `toggle`.
- **Urgent** The client's urgency state. When `on`, the client's layout box will be highlighted. Possible values are `on`, `off`, and `toggle`.
- **kill** When written, the window is closed politely, if possible.
- **slay** When written, the client is killed peremptorily.

props The client's window class (the X11 `WM_CLASS` property) and title string, separated by colons. This file is not writable.

label The client's window title. May be written to change the client's title.

tags The client's tags. Tag names are separated by `+` signs. Tags beginning and ending with `/` are treated as regular expressions. If the written value begins with a `+` or a `-`, the tags are updated rather than overwritten. Tag names which directly follow a `-` sign are removed rather than added. Regular expression tags which directly follow a minus sign are treated as exclusion expressions. For example, the tag string `+/foo/-/food/` will match the tag `foobar`, but not the tag `foodstand`.

3.6.1 Key Bindings

To control clients, we'll add the following key bindings:

```
⟨⟨Client Command Keys⟩⟩ ::=
Mod4-Shift-c) wmiir xwrite /client/sel/ctl kill;;
Mod4-f) wmiir xwrite /client/sel/ctl Fullscreen toggle;;
```

And to manage their tags, we'll need:

```
⟨⟨Tagging Keys⟩⟩ ::=
Mod4-Shift-t)
  # Get the selected client's id
  c=$(wmiir read /client/sel/tag | sed 1q)
  # Prompt the user for new tags
  tags=$(wmiir ls /tag | sed 's,/,,; /sel/d' | wimenu)
  # Write them to the client
  wmiir xwrite /client/$c/tags $tag;;
Mod4-Shift-[0-9])
  wmiir xwrite /client/sel/tags ${2##*-};;
```

3.6.2 Click Menus

```
⟨⟨Client Menu Events⟩⟩ ::=
ClientMouseDown) # ClientMouseDown ⟨Client ID⟩ ⟨Button⟩
  [ $2 = 3 ] && clickmenu \
    "Delete:xwrite /client/$1/ctl kill" \
    "Kill:xwrite /client/$1/ctl slay" \
    "Fullscreen:/client/$1/ctl Fullscreen on"
```

3.6.3 Unresponsive Clients

When wmii tries to close a window, it waits 8 seconds for the client to respond, and then lets its scripts decide what to do with it. The stock scripts prompt the user for input:

```
⟨⟨Unresponsive Clients⟩⟩ ::=
UnresponsiveClient) # UnresponsiveClient ⟨Client ID⟩
{
    # Use wihack to make the xmessage a transient window of
    # the problem client. This will force it to open in the
    # floaing layer of whatever views the client is attached to
    resp=$(wihack -transient $1 \
           xmessage -nearmouse -buttons Kill,Wait -print \
           "The following client is not responding." \
           "What would you like to do?$(echo)" \
           $(wmiir read /client/$1/label))
    [ $resp = Kill ] && wmiir xwrite /client/$1/ctl slay
} &;;
```

3.7 Views

Views are represented by directories under the /tag/ tree. The special sel/ directory represents the currently selected client. The sel tag is treated similarly elsewhere. The files in these directories are:

ctl The view's control file. The properties are:

select ⟨***Area***⟩ Select the column ⟨*Area*⟩, where ⟨*Area*⟩ is a 1-based column index, or ~ for the floating area.

select ⟨***Area***⟩ ⟨***Client Index***⟩ Select the column ⟨*Area*⟩, and the ⟨*Client Index*⟩th client.

select client ⟨***Client ID***⟩ Select the client with the X11 window ID ⟨*Client ID*⟩.

select ⟨***Direction***⟩ Select the client in ⟨*Direction*⟩ where ⟨*Direction*⟩ may be one of ⟨*up* ∧ *down* ∧ *left* ∧ *right*⟩.

send client ⟨***Client ID***⟩ ⟨***Area***⟩ Send ⟨*Client ID*⟩ to ⟨*Area*⟩. ⟨*Area*⟩ may be `sel` for the selected area, and `client` ⟨`Client ID`⟩ may be `sel` for the currently selected client.

send client ⟨***Client ID***⟩ ⟨***Direction***⟩ Send ⟨*Client ID*⟩ to a column or position in its column in the given direction.

send client ⟨***Client ID***⟩ **toggle** If ⟨*Client ID*⟩ is floating, send it to the managed layer. If it's managed, send it to the floating layer.

swap client ⟨***Client ID***⟩ **...** The same as the `send` commands, but swap ⟨*Client ID*⟩ with the client at the given location.

colmode ⟨***Area***⟩ ⟨***Mode***⟩ Set ⟨*Area*⟩'s mode to ⟨*Mode*⟩, where ⟨*Mode*⟩ is a string of values similar to tag specifications. Values which may be added and removed are as follows for managed areas:

stack One and only one client in the area is uncollapsed at any given time. When a new client is selected, it is uncollapsed and the previously selected client is collapsed.

max Collapsed clients are hidden from view entirely. Uncollapsed clients display an indicator ⟨*n*⟩/⟨*m*⟩, where ⟨*m*⟩ is the number of collapsed clients directly above and below the client, plus one, and ⟨*n*⟩ is the client's index in the stack.

For the floating area, the values are the same, except that in `max` mode, floating clients are hidden when the managed layer is selected.

grow ⟨***Frame***⟩ ⟨***Direction***⟩ **[**⟨***Amount***⟩**]** Grow ⟨*Frame*⟩ in the given direction, by ⟨*Amount*⟩. ⟨*Amount*⟩ may be any integer, positive or negative. If suffixed with `px`, it specifies an exact pixel amount, otherwise it specifies a "reasonable increment". Defaults to 1.

⟨*Frame*⟩ may be one of:

- client ⟨*Client ID*⟩
- ⟨*Area*⟩ ⟨*Client Index*⟩

nudge ⟨***Frame***⟩ ⟨***Direction***⟩ **[**⟨***Amount***⟩**]** The same as `grow`, but move the client in ⟨*Direction*⟩ instead of resizing it.

3.7.1 Key Bindings

We'll use the following key bindings to interact with views:

```
⟨⟨Motion Keys⟩⟩ ::=
Mod4-h) wmiir xwrite /tag/sel/ctl select left;;
Mod4-l) wmiir xwrite /tag/sel/ctl select right;;
Mod4-k) wmiir xwrite /tag/sel/ctl select up;;
Mod4-j) wmiir xwrite /tag/sel/ctl select down;;
Mod4-space) wmiir xwrite /tag/sel/ctl select toggle;;

⟨⟨Client Movement Keys⟩⟩ ::=
Mod4-Shift-h) wmiir xwrite /tag/sel/ctl send sel left;;
Mod4-Shift-l) wmiir xwrite /tag/sel/ctl send sel right;;
Mod4-Shift-k) wmiir xwrite /tag/sel/ctl send sel up;;
Mod4-Shift-j) wmiir xwrite /tag/sel/ctl send sel down;;
Mod4-Shift-space) wmiir xwrite /tag/sel/ctl send sel toggle;;

⟨⟨Column Mode Keys⟩⟩ ::=
Mod4-d) wmiir xwrite /tag/sel/ctl colmode sel -stack-max;;
Mod4-s) wmiir xwrite /tag/sel/ctl colmode sel stack-max;;
Mod4-m) wmiir xwrite /tag/sel/ctl colmode sel stack+max;;
```

3.7.2 Click Menus

```
⟨⟨Tag Menu Events⟩⟩ ::=
LeftBarMouseDown) # LeftBarMouseDown ⟨Button⟩ ⟨Bar Name⟩
  [ $1 = 3 ] && clickmenu \
    "Delete:delete_view $2"
```

3.8 Command and Program Execution

Perhaps the most important function we need to provide for is the execution of programs. Since `wmii` users tend to use terminals often, we'll add a direct shortcut to launch one. Aside from that, we'll add a menu to launch arbitrary programs (with completions) and a separate menu to launch wmii specific commands.

We use `wmiir setsid` to launch programs with their own session IDs to prevent untoward effects when this script dies.

```
⟨⟨Command Execution Initialization⟩⟩ ::=
terminal() { wmiir setsid xterm "$@" }
proglist() {
    IFS=: set -- $1
    find -L $@ -maxdepth 1 -perm /111 | sed 's,.*/,,' | sort | uniq
    unset IFS
}
```

3.8.1 Key Bindings

```
⟨⟨Command Execution Keys⟩⟩ ::=
Mod4-Return) terminal & ;;
Mod4-p) eval exec wmiir setsid "$(proglist $PATH | wimenu)" &;;
```

```
Mod4-a) {
    set -- $(proglist $WMII_CONFPATH | wimenu)
    prog = $( (PATH=$WMII_CONFPATH which $1) ); shift
    eval exec $prog "$@"
} &;;
```

3.9 The Root

The root filesystem contains the following:

ctl The control file. The properties are:

bar on ⟨***top*** ∧ ***bottom***⟩ Controls where the bar is shown.

bar off Disables the bar entirely.

border The border width, in pixels, of floating clients.

colmode ⟨***Mode***⟩ The default column mode for newly created columns.

focuscolors ⟨***Color Tuple***⟩ The colors of focused clients.

normcolors ⟨***Color Tuple***⟩ The colors of unfocused clients and the default color of bar buttons.

font ⟨***Font***⟩ The font used throughout `wmii`. If prefixed with `xft:`, the Xft font renderer is used, and fonts may be antialiased.

grabmod ⟨***Modifier Keys***⟩ The key which must be pressed to move and resize windows with the mouse without clicking hot spots.

incmode ⟨***Mode***⟩ Controls how X11 increment hints are handled in managed mode. Possible values are:

ignore Increment hints are ignored entirely. Clients are stretched to fill their full allocated space.

show Gaps are shown around managed client windows when their increment hints prevent them from filling their entire allocated space.

squeeze When increment hints cause gaps to show around clients, `wmii` will try to adjust the sizes of the clients in the column to minimize lost space.

view ⟨***Tag***⟩ The currently visible view.

exec ⟨***Command***⟩ Replaces this `wmii` instance with ⟨*Command*⟩. ⟨*Command*⟩ is split according to rc quoting rules, and no expansion occurs. If the command fails to execute, `wmii` will respawn.

spawn ⟨***Command***⟩ Spawns ⟨*Command*⟩ as it would spawn `wmiirc` at startup. If ⟨*Command*⟩ is a single argument and doesn't begin with `/` or `./`, `$WMII_CONFPATH` is searched for the executable. Otherwise, the whole argument is passed to the shell for evaluation.

props The client's window class (the X11 `WM_CLASS` property) and title string, separated by colons. This file is not writable.

label The client's window title. May be written to change the client's title.

tags The client's tags. Tag names are separated by + signs. Tags beginning and ending with / are treated as regular expressions. If the written value begins with a + or a -, the tags are updated rather than overwritten. Tag names which directly follow a - sign are removed rather than added. Regular expression tags which directly follow a minus sign are treated as exclusion expressions. For example, the tag string +/foo/-/food/ will match the tag foobar, but not the tag foodstand.

3.9.1 Configuration

We'll need to write our previously defined theme information to wmii:

```
⟨⟨Configuration⟩⟩ ::=
⟨⟨Theme Definitions⟩⟩

xsetroot -solid $background
wmiir write /ctl <<!
border 2
focuscolors $focuscolors
normcolors $normcolors
font $font
grabmod Mod4
!
```

3.9.2 Key Bindings

And we need a few more key bindings to select our views:

```
⟨⟨Tag Selection Keys⟩⟩ ::=
Mod4-Shift-t)
  # Prompt the user for a tag
  tags=$(wmiir ls /tag | sed 's,/,,; /sel/d' | wimenu)
  # Write it to the filesystem.
  wmiir xwrite /ctl view $tag;;
Mod4-[0-9])
  wmiir xwrite /ctl view ${2##*-};;
```

3.10 Tieing it All Together

```
#!/bin/sh
⟨⟨Click Menu Initialization⟩⟩
⟨⟨Command Execution Initialization⟩⟩

⟨⟨Configuration⟩⟩

⟨⟨Bind Keys⟩⟩
⟨⟨Event Loop⟩⟩
```

3.11 The End Result

For clarity, here is the end result:

```
#!/bin/sh
# ⟨⟨Click Menu Initialization⟩⟩
clickmenu() {
    if res=$(wmii9menu -- "$@"); then eval "$res"; fi
}
# ⟨⟨Command Execution Initialization⟩⟩
terminal() { wmiir setsid xterm "$@" }
proglist() {
    IFS=: set -- $1
    find -L $@ -maxdepth 1 -perm /111 | sed 's,.*/,,' | sort | uniq
    unset IFS
}

# ⟨⟨Configuration⟩⟩
# ⟨⟨Theme Definitions⟩⟩
normcolors='#000000 #c1c48b #81654f'
focuscolors='#000000 #81654f #000000'
background='#333333'
font='drift,-*-fixed-*-*-*-*-9-*-*-*-*-*-*-*-*'

xsetroot -solid $background
wmiir write /ctl <<!
border 2
focuscolors $focuscolors
normcolors $normcolors
font $font
grabmod Mod4
!

# ⟨⟨Bind Keys⟩⟩
{
cat <<!
Mod4-space
Mod4-d
Mod4-s
Mod4-m
Mod4-a
Mod4-p
Mod4-t
Mod4-Return
Mod4-Shift-space
Mod4-f
Mod4-Shift-c
Mod4-Shift-t
```

```
Mod4-h
Mod4-j
Mod4-k
Mod4-l
Mod4-Shift-h
Mod4-Shift-j
Mod4-Shift-k
Mod4-Shift-l
!
for i in 1 2 3 4 5 6 7 8 9 0; do
    echo Mod4-$i
    echo Mod4-Shift-$i
done
} | wmiir write /keys

# ⟨⟨Event Loop⟩⟩
# Broadcast a custom event
wmiir xwrite /event Start wmiirc

# Turn off globbing
set -f
# Open /event for reading
wmiir read /event |
# Read the events line by line
while read line; do
    # Split the line into words, store in $@
    set -- $line
    event=$1; shift
    line = "$(echo $line | sed 's/^[^ ]* //' | tr -d '\n')"

    # Process the event
    case $event in
    Start) # Quit when a new instance starts
        [ $1 = wmiirc ] && exit;;

    # ⟨⟨Event Handlers⟩⟩
    # ⟨⟨View Button Events⟩⟩
    CreateTag)  # CreateTag ⟨Tag Name⟩
        echo $normcolors $1 | wmiir create /lbar/$1;;
    DestroyTag) # DestroyTag ⟨Tag Name⟩
        wmiir rm /lbar/$1;;
    FocusTag)   # FocusTag ⟨Tag Name⟩
        wmiir xwrite /lbar/$1 $focuscolors $1;;
    UnfocusTag) # UnfocusTag ⟨Tag Name⟩
        wmiir xwrite /lbar/$* $normcolors $1;;

    # ⟨⟨Urgency Events⟩⟩
    # The urgency events are 'Client' events when the program
```

```
# owning the window sets its urgency state. They're 'Manager'
# events when wmii or the wmii user sets the state.
UrgentTag)    # UrgentTag ⟨'Client' or 'Manager'⟩ ⟨Tag Name⟩
    wmiir xwrite /lbar/$2 *$2;;
NotUrgentTag) # NotUrgentTag ⟨'Client' or 'Manager'⟩ ⟨Tag Name⟩
    wmiir xwrite /lbar/$2 $2;;

# ⟨⟨Unresponsive Clients⟩⟩
UnresponsiveClient) # UnresponsiveClient ⟨Client ID⟩
{
    # Use wihack to make the xmessage a transient window of
    # the problem client. This will force it to open in the
    # floaing layer of whatever views the client is attached to
    resp=$(wihack -transient $1 \
           xmessage -nearmouse -buttons Kill,Wait -print \
           "The following client is not responding." \
           "What would you like to do?$(echo)" \
           $(wmiir read /client/$1/label))
    [ $resp = Kill ] && wmiir xwrite /client/$1/ctl slay
} &;;

# ⟨⟨Notice Events⟩⟩
Notice)
    wmiir xwrite /rbar/!notice $line
    kill $xpid 2>/dev/null # Let's hope this isn't reused...
    { sleep 5; wmiir xwrite /rbar/!notice ' '; } &
    xpid = $!;;

# ⟨⟨Key Events⟩⟩
Key) # Key ⟨Key Name⟩
    case $1 in
    # ⟨⟨Motion Keys⟩⟩
    Mod4-h) wmiir xwrite /tag/sel/ctl select left;;
    Mod4-l) wmiir xwrite /tag/sel/ctl select right;;
    Mod4-k) wmiir xwrite /tag/sel/ctl select up;;
    Mod4-j) wmiir xwrite /tag/sel/ctl select down;;
    Mod4-space) wmiir xwrite /tag/sel/ctl select toggle;;

    # ⟨⟨Client Movement Keys⟩⟩
    Mod4-Shift-h) wmiir xwrite /tag/sel/ctl send sel left;;
    Mod4-Shift-l) wmiir xwrite /tag/sel/ctl send sel right;;
    Mod4-Shift-k) wmiir xwrite /tag/sel/ctl send sel up;;
    Mod4-Shift-j) wmiir xwrite /tag/sel/ctl send sel down;;
    Mod4-Shift-space) wmiir xwrite /tag/sel/ctl send sel toggle;;

    # ⟨⟨Column Mode Keys⟩⟩
    Mod4-d) wmiir xwrite /tag/sel/ctl colmode sel -stack-max;;
    Mod4-s) wmiir xwrite /tag/sel/ctl colmode sel stack-max;;
```

```
    Mod4-m) wmiir xwrite /tag/sel/ctl colmode sel stack+max;;

    # ⟨⟨Client Command Keys⟩⟩
    Mod4-Shift-c) wmiir xwrite /client/sel/ctl kill;;
    Mod4-f) wmiir xwrite /client/sel/ctl Fullscreen toggle;;

    # ⟨⟨Command Execution Keys⟩⟩
    Mod4-Return) terminal & ;;
    Mod4-p) eval exec wmiir setsid "$(proglist $PATH | wimenu)" &;;
    Mod4-a) {
        set -- $(proglist $WMII_CONFPATH | wimenu)
        prog = $( (PATH=$WMII_CONFPATH which $1) ); shift
        eval exec $prog "$@"
    } &;;

    # ⟨⟨Tag Selection Keys⟩⟩
    Mod4-Shift-t)
      # Prompt the user for a tag
      tags=$(wmiir ls /tag | sed 's,/,,; /sel/d' | wimenu)
      # Write it to the filesystem.
      wmiir xwrite /ctl view $tag;;
    Mod4-[0-9])
      wmiir xwrite /ctl view ${2##*-};;

    # ⟨⟨Tagging Keys⟩⟩
    Mod4-Shift-t)
      # Get the selected client's id
      c=$(wmiir read /client/sel/tag | sed 1q)
      # Prompt the user for new tags
      tags=$(wmiir ls /tag | sed 's,/,,; /sel/d' | wimenu)
      # Write them to the client
      wmiir xwrite /client/$c/tags $tag;;
    Mod4-Shift-[0-9])
      wmiir xwrite /client/sel/tags ${2##*-};;

    esac;;

# ⟨⟨Client Menu Events⟩⟩
ClientMouseDown) # ClientMouseDown ⟨Client ID⟩ ⟨Button⟩
  [ $2 = 3 ] && clickmenu \
    "Delete:xwrite /client/$1/ctl kill" \
    "Kill:xwrite /client/$1/ctl slay" \
    "Fullscreen:/client/$1/ctl Fullscreen on"

# ⟨⟨Tag Menu Events⟩⟩
LeftBarMouseDown) # LeftBarMouseDown ⟨Button⟩ ⟨Bar Name⟩
  [ $1 = 3 ] && clickmenu \
    "Delete:delete_view $2"
```

```
    esac
done
```

Index