PLWM – The Pointless Window Manager

Peter Liljenberg

Copyright © 1999-2002 Peter Liljenberg

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled “Copying” and “GNU General Public License” are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

Table of Contents

Introduction 1

1 Philosophy and Excuses 2

2 Running PLWM 3

3 Configuration 5

4 Core Classes 6
- 4.1 `WindowManager` Public Attributes 6
- 4.2 `Screen` Public Attributes 7
- 4.3 `Client` Public Attributes 7

5 Event Handling 9
- 5.1 Event Objects 9
- 5.2 EventFetcher 9
- 5.3 EventDispatcher 11

6 Core Events 13

7 Client Filters 14

8 Extension Modules 16
- 8.1 Extension Coding Conventions 16
- 8.2 Extension Classes Initialization 16
- 8.3 Available Extension Modules 17
  - 8.3.1 `color` Extension Module 17
  - 8.3.2 `font` Extension Module 17
  - 8.3.3 `keys` Extension Module 18
  - 8.3.4 `focus` Extension Module 19
  - 8.3.5 `border` Extension Module 20
  - 8.3.6 `outline` Extension Module 20
  - 8.3.7 `moveresize` Extension Module 21
  - 8.3.8 `cycle` Extension Module 22
  - 8.3.9 `views` Extension Module 24
  - 8.3.10 `menu` Extension Module 25
  - 8.3.11 `panes` Extension Module 26
  - 8.3.12 `modewindow` Extension Module 28
  - 8.3.13 `modestatus` Extension Module 29
  - 8.3.14 `mw_clock` Extension Module 30

8.3.15 `mw_biff` Extension Module 30
8.3.16 `mw_apm` Extension Module 31
8.3.17 `input` Extensions Module 31
8.3.18 `inspect` Extension Module 33

9 Utilities 34
9.1 wmm 34
9.2 inspect_plwm 34

10 ToDo 36

11 Credits 37

12 Contact Info 38

Introduction

PLWM, The Pointless Window Manager, is a collection of window manager primitives written in Python.

If you are just interested in running PLWM you should at least read the chapter on the underlying philosophy, take a look at the chapter on running PLWM and have `README.examplewm` at hand.

If you are interested in really using PLWM it is recommended, even required, to read the rest of the manual.

1 Philosophy and Excuses

PLWM is not a normal window manager, in fact, it isn't a window manager at all. Instead it is a collection of Python classes which you can use to build your own window manager. You can include the existing features you like and easily write your own extensions to make your PLWM behave exactly as you want it to. Eventually, you will have a perfect symbiosis of user and window manager, you and the computer will be a beautiful Mensch-Maschine!

One of the basic ideas is that the mouse should be banished, and everything should be possible to do without moving your hands from the keyboard. This is the pointless bit of PLWM.

Another of the other basic ideas is to make a window manager which is is pure Unix Philosophy: a bunch of simple tools which can be combined to build a powerful application. The `"tools"` are Python classes which makes it easy to inherit, extend, mixin and override functionality to get exactly the behaviour that you want.

This makes PLWM extremely configurable by sacrificing ease of configuration: you actually have to write some Python code to get the window manager exactly as you want it. However, if you was moved by the first paragraph, then you're probably already a hacker and will relish in writing your own window manager.

A typical PLWM setup might look rudimentary, even hostile, to people used to the glitz and glamour of more conventional window managers. However, there are a lot of powerful features, making it really user-friendly. Provided that the user is friendly to PLWM, of course.

2 Running PLWM

PLWM, at least in the `examplewm.py` guise, is not very sophisticated when it comes to command line arguments. It only groks three:

`-display`	Run PLWM against another display than `$DISPLAY`.
`-debug`	Enable debug messages. The argument should be a comma-separated list of debug message categories, or empty, meaning enable all messages.
`-version`	Print the PLWM version.

It can be tricky at first to figure out how to have a hacking-friendly X setup, so here is a fragment of my '`.xinitrc`' as an example of a PLWM environment:

```
#!/bin/sh

# Redirect error messages to a log file.  This is where PLWM
# tracebacks will go, so keep an eye on it.
exec 1>$HOME/.x11startlog 2>$HOME/.x11startlog

# Read resource database from file .Xdefaults
xrdb ~/.Xdefaults

# Set a solid color for the root window.  The modewindow will have the
# same background (set with .Xdefaults) and no border, so it will appear
# to be a part of the root.  Good enough.

xsetroot -solid darkolivegreen

# Desperately try to start some window manager
# As we start it in the background it can exit without shutting down
# the entire X server.

(plwm || ctwm || twm) &

# Instead the X server is kept running by wmm, or if that fails, by
# xlogo. To shut down the X server we kill the wmm or xlogo window.

wmm || xlogo
```

How to configure WMM (see Section 9.1 [wmm], page 34) is not obvious at first either, so here's a '`.wmmrc`' too (notice the tabs between the columns):

```
plwm	plwm &
examplewm	/home/petli/hack/plwm/examples/examplewm.py &
twm	twm &
```

The idea is to use a stable PLWM installed in your `$PATH` by default. When you are about to test some freshly hacked feature or a bugfix, simply kill off the running PLWM (C-M-Esc in `examplewm.py`). This will pop up the WMM window, so click on the examplewm button in it to start the development version, using modules from `plwm` directories next to `examples`.

To put the finishing touches to the configuration, we can change some fonts and colors with the '`~/.Xdefaults`' file:

```
Plwm.outline.font: -*-lucida-bold-r-*-sans-20-*-*-*-*-*-*-*
Plwm.border.color: black
Plwm.border.focus.color: grey60

Plwm.modewindow.background: darkolivegreen
Plwm.modewindow.foreground: white
```

3 Configuration

Most of the configuration of PLWM is done by writing a python script using the various window manager modules. See Chapter 8 [Extension Modules], page 16.

However, some small parts of PLWM can be configured with X resources. The various X resources are defined in the section of their corresponding modules.

They have one thing in common though, in that they all start with the name component `plwm` and the class component `Plwm`. The name component used in lookups will actually be the name of the window manager script, so if your script is called `foowm.py`, the resources looked up will be e.g. `foowm.border.color` instead of `plwm.border.color`. The class component used is controlled by the `WindowManager` attribute `appclass`, which by default is `Plwm`.

Summary: in your '`~/.Xdefaults`' or '`~/.Xresources`', start all PLWM resources with `Plwm.`, unless you know what you're doing.

4 Core Classes

If you absolutely must point to a single module and call it PLWM, then you should direct your attention to `wmanager`. It contains three classes which implements the fundamental window managing and provides the extension framework.

WindowManager	This is the central window manager class. A `WindowManager` instance manages the windows on a single display, which is provided as an argument to the constructor. `WindowManager` also drives the event handling loop.
Screen	This class takes care of catching client windows so we can manage them. When a `WindowManager` instance is created it will create a `Screen` instance for each physical screen (monitor) connected to the display. Normally, an X server only have one screen, but they can also be multiheaded, i.e. having more than one screen. Each screen has its own root window, and all windows are therefore local to a certain screen and cannot move between different screens.
Client	This class manages a single window. Instances are created when the `Screen` instance managing the screen detects that a window has been created. All window operations by extensions should be done using methods on the `Client` instance managing that window.

This gives us the following structure: A single `WindowManager` instance manages a certain display, by managing one or more `Screen` instances which in turn manages a number of `Client` instances.

These classes have a number of publicly available attributes, listed in the sections below. Of course, all member attributes are available for you, but if you try to stick to the listed attributes and use the methods provided by the classes to manipulate windows your code will stand a better chance of working with future extensions.

The core classes also generates some events which can be useful for mixins. See Chapter 6 [Core Events], page 13.

4.1 `WindowManager` Public Attributes

display [Instance Variable of WindowManager]

A `Xlib.display.Display` object connecting us to the X server.

events [Instance Variable of WindowManager]

An `EventFetcher` object for this display. See Chapter 5 [Event Handling], page 9.

dispatch [Instance Variable of WindowManager]

A global `EventDispatcher`. See Chapter 5 [Event Handling], page 9.

current_client [Instance Variable of WindowManager]
focus_client [Instance Variable of WindowManager]

The client currently containing the pointer and the client which has keyboard focus, respectivly. Most of the time these are the same, but certain windows do not use input and will therefore never be focused. Most operations should be performed on current_client. The `WindowManager` provides the method `set_current_client()` to change this in the proper way. However, to implement some kind of focus scheme you have to use some extension class, see Section 8.3.4 [focus], page 20.

screens [Instance Variable of WindowManager]

A list of the managed screens.

screen_nums [Instance Variable of WindowManager]
screen_roots [Instance Variable of WindowManager]

Mappings from screen numbers or root windows to the corresponding screen object.

default_screen [Instance Variable of WindowManager]

The default screen, defined when opening the display.

current_screen [Instance Variable of WindowManager]

The screen currently containing the pointer. This will be maintained by the screens automatically.

4.2 `Screen` Public Attributes

wm [Instance Variable of Screen]

The `WindowManager` which holds this screen.

number [Instance Variable of Screen]

The number of this screen.

root [Instance Variable of Screen]

The root window of this screen.

dispatch [Instance Variable of Screen]

The `EventDispatcher` for the root window. See Chapter 5 [Event Handling], page 9.

info [Instance Variable of Screen]

The screen information structure, as returned by the Xlib `Display` method `screen()`.

4.3 `Client` Public Attributes

screen [Instance Variable of Client]
wm [Instance Variable of Client]

The `Screen` and `WindowManager` instances which contains this client.

withdrawn [Instance Variable of Client]

Set to true if this client has been withdrawn. A withdrawn client should not be modified further, and has already been removed from the `Screen`'s list of clients.

dispatch [Instance Variable of Client]

The `EventDispatcher` for the client window. See Chapter 5 [Event Handling], page 9.

current [Instance Variable of Client]
focused [Instance Variable of Client]

These attributes will be true or false to indicate whether this client is the current one, and if it has focus.

5 Event Handling

Event handling consists of getting events and then distributing them to the event handlers. In PLWM this is handled by the module `event` and its classes `EventFetcher` and `EventDispatcher`. If the event loop in the `WindowManager` is the heart driving the event blood stream, `EventFetcher` is the lungs providing fresh events (the oxygene) and `EventDispatcher` is the arteries delivering the events to all the working parts of the window manager body.

No, that analogue wasn't strictly necessary.

5.1 Event Objects

The events are Python objects, either X event objects from Python Xlib, or an instance of some other Python class. The only thing required by the event system is that they have at least this attribute:

type [Instance Variable of Event Objects]

This identifies the event type, and can be any hashable object. For X events this is some integer constants defined in `Xlib.X`. For other events this can be the event object class, a unique string, or anything else that is useful. It must however be a hashable object, since the type is used as an index into dictionaries.

The `event` module provides a function for getting new, unique integer event types:

new_event_type ( ) [Function]

Return a new unique integer event type, which does not conflict with the event types of the Xlib.

Additionally the `WindowManager` uses one of these attributes to figure out which `Screen` and `Client` that should take care of the event:

client [Instance Variable of Event Objects]

A client object this event is about or for. The `Screen` managing this client will get the event, and can then in its turn pass the event on to the client itself.

window [Instance Variable of Event Objects]

The window object this event is about or for. The `Screen` managing the root window of this window will get the event. If the window corresponds to a managed client, that client can also get the event.

screen [Instance Variable of Event Objects]

The screen object this event is about or for, which will get the event.

If the event has none of the attributes or it is for an unmanaged screen it will only be passed to the global event handlers.

5.2 EventFetcher

The `EventFetcher` can provide events to the window manager from a number of sources: the X server, timers, files or the window manager itself.

Synthetic events are generated by some code in the window manager, typically as an abstraction of some user input. As an example, the `focus` module generates `ClientFocusOut` and `ClientFocusIn` events when the focus change, which it can do as an effect of an `X.EnterNotify` event, or a call to `focus.move_focus`. Synthetic events have precedence over all other event types.

put_event (*event*) [Method on `EventFetcher`]

Insert a synthetic event object. Synthetic events are always returned before any timer or X events, in FIFO order.

Timer events are used to do something at a later time. The `keys` module uses it to implement a time-out on keyboard grabs, and the `mw_clock` module uses it to update a clock display once a minute. Timer events are represented by `TimerEvent` objects, and have precedence over file and X events. A timer event object is not reusable, so if something should be done periodically, a new timer event will have to be rescheduled whenever the previous one expires.

add_timer (*timer*) [Method on `EventFetcher`]

Add the `TimerEvent` object *timer* to the list of timers. The expiration time of the event is specified when creating the timer event. The event will be returned when the timer expires unless it is cancelled before that.

TimerEvent (*event_type, after = 0, at = 0*) [Class]

Create a a timer event that will expire either at a relative time, set with *after*, or at a specific time, set with *at*. Times are measured in seconds as in the `time` module, and can be integers or floating point values.

Timer events are identified by its `type` member, which are specified with the *event_type* argument.

cancel ( ) [Method on `TimerEvent`]

Cancel this timer event, if it hasn't expired yet.

File events can be used to handle non-blocking I/O. They are generated when some file is ready for reading or writing, and is typically used for network services, e.g. by the `inspect` module. File events are represented by `FileEvent` objects, which remain on the list of watched files until they are explicitly cancelled. File events have the lowest priority, and are preceeded by X events.

add_file (*file*) [Method on `EventFetcher`]

Add the `FileEvent` object *file* to the list of watched files. It will be returned whenever its file is ready for the choosen I/O operation.

FileEvent (*event_type, file, mode = None*) [Class]

Create a file event wathing *file*, which could be any object with a `fileno()` method. The event is identified by *event_type*.

mode is the types of I/O that the caller is interested in, and should be a bitmask of the flags `FileEvent.READ`, `FileEvent.WRITE`, or `FileEvent.EXCEPTION`. If *mode* is `None`, the `mode` attribute of *file*, as specified to the `open()` call, will be used instead.

state [Instance Variable of FileEvent]

When a file event is returned by the event loop, this attribute will be set to a mask of the I/O modes that the file is ready to perform, a subset of the modes waited for.

If `FileEvent.READ` is set, at least one byte can be read from the file without blocking. If `FileEvent.WRITE` is set, at least one byte can be written to the file without blocking. If `FileEvent.EXCEPTION` is set, some exceptional I/O has occured, e.g. out-of-band data on a TCP socket.

set_mode (*newmode = None, set = 0, clear = 0*) [Method on `FileEvent`]

Change the I/O modes waited for. If *newmode* is not `None`, the mode will be reset to *newmode*, otherwise the old mode will be modifed. Then the flags in the bitmask *set* will be added, and the flags in *clear* will be removed, in that order.

cancel ( ) [Method on `FileEvent`]

Cancel this file event, removing it from the list of watched files.

5.3 EventDispatcher

Each `Screen` and `Client` has an `EventDispatcher` connected to their root window or client window, respectivly. Additionally, the `WindowManager` has an `EventDispatcher` which is connected to all the root windows (through the dispatchers of each `Screen`).

An event is passed to the event handling functions which have been registered for that particular event type. There are three levels of event handlers in a dispatcher:

System Handlers

System handlers will always be called, even if grab handlers are installed. They will be called before the other types of handlers in this dispatcher. They are primarily meant to be used by the core classes.

Grab Handlers

Grab handlers override previously installed grab handlers and all normal handlers, but not system handlers.

Normal Handlers

Normal handlers are the most useful type of handlers for extension modules. They will be called only if there are no grab handlers installed for this event type.

First of all the handlers in the global dispatcher are called. Then, if the event can be associated with a managed screen through its `client`, `window`, or `screen` attributes the handlers in the dispatcher for that screen are called. Finally, if the event is for a managed window the handlers in the dispatcher for that client are called.

Grab handlers do not interfere with the dispatcher sequence directly, but a grab handler do block grab handlers and normal handlers in later dispatchers. System handler are always called, though.

An example: Assume that an event for a managed client has been fetched and is about to be passed through the dispatchers. The matching event handlers are the following: in the global dispatcher one system handler and two normal handlers, in the screen dispatcher a grab handler and one normal handler, and in the client dispatcher one system handler, one grab handler and one normal handler. The handlers will be called in this order: global system, global normals, screen grab, client system. The screen normal, client grab and normal handlers will be ignored because of the grab handler in the screen.

Handlers are registered with one of these methods:

add_handler (*type, handler,* [*masks = None*], [*handler_id = None*]) [Method on `EventDispatcher`]

add_grab_handler (*type, handler,* [*masks = None*], [*handler_id = None*]) [Method on `EventDispatcher`]

add_system_handler (*type, handler,* [*masks = None*], [*handler_id = None*]) [Method on `EventDispatcher`]

Add a handler for *type* events. *handler* is a function which will get one argument, the event object.

If *masks* is omitted or None the default X event masks for the event type will be set for the `EventDispatcher`'s window. Otherwise it should be an event mask or a list or tuple of event masks to set.

handler_id identifies this handler, and defaults to the *handler* itself if not provided.

remove_handler (*handler_id*) [Method on `EventDispatcher`]

Remove the handler or handlers identified by *handler_id*. This will also clear the masks the handlers had installed.

Event masks can also be handled manually when necessary. All event masks keep a reference count, so calls to the following functions nest neatly.

set_masks (*masks*) [Method on `EventDispatcher`]

Set *masks* on the window, without installing any handlers. *masks* should be an event mask or a list or tuple of event masks to set.

unset_masks (*masks*) [Method on `EventDispatcher`]

Clear *masks* on the window. *masks* should be an event mask or a list or tuple of event masks to set.

block_masks (*masks*) [Method on `EventDispatcher`]

Block *masks* on the window. This will prevent any matching X events to be generated on the window until a matching unblock_masks. *masks* should be an event mask or a list or tuple of event masks to set.

unblock_masks (*masks*) [Method on `EventDispatcher`]

Unblock *masks* on the window, allowing the matching X events to be generated. *masks* should be an event mask or a list or tuple of event masks to set.

6 Core Events

The core classes can generate a number of internal events. Mostly, these are a result of some X event or user activity. All of these events are defined in the module `plwm.wmevents`.

AddClient [Event]

Generated when a client is added. The added client is identified by the event object attribute `client`.

RemoveClient [Event]

Generated when a client is removed (withdrawn). The removed client is identified by the event object attribute `client`.

QuitWindowManager [Event]

Generated when the window manager event loop is exited by calling `WindowManager.quit`.

CurrentClientChange [Event]

Generated when a new client is made current. The event object has two attributes: `client` is the new client, or `None` if no client is current now. `screen` is the screen of the previous current client, or `None` if no client was current previously.

ClientFocusOut [Event]

Generated when a client loses focus. The event object attribute `client` is the now unfocused client.

ClientFocusIn [Event]

Generated when a client gets focus. The event object attribute `client` is the now focused client.

7 Client Filters

The module `plwm.cfilter` defines a number of client filters. These filters can be called with a client as an argument, and returns true if the client matches the filter, or false otherwise. Extension modules can then provide customization with client filters, allowing the user to give certain clients special treatment.

Currently, the following filters are defined:

true [Filter]
all [Filter]

These filters are true for all clients.

false [Filter]
none [Filter]

These filters are false for all clients.

is_client [Filter]

True if the object is a `wmanager.Client` instance.

iconified [Filter]

True if the client is iconified.

mapped [Filter]

True if the client is mapped, the opposite of iconified.

name (*string*) [Filter]
re_name (*regexp*) [Filter]
glob_name (*pattern*) [Filter]

These filters are true if the client resource name or class is exactly *STRING*, or matches the regular expression *regexp* or the glob pattern *pattern*.

title (*string*) [Filter]
re_title (*regexp*) [Filter]
glob_title (*pattern*) [Filter]

These filters are similar to the name filters above, but matches the client title instead.

These basic filters can then be assembled into larger, more complex filters using the following logical operations:

And (*filter1, filter2, ..., filterN*) [Filter]

True if all of the subfilters are true.

Or (*filter1, filter2, ..., filterN*) [Filter]

True if at least one of the subfilters is true.

Not (*filter*) [Filter]

True if *filter* is false.

Here are some examples of compound filters:

```
# Match any client that isn’t an Emacs

Not(name(’Emacs’))


# Match an iconified xterm:

And(iconified, name(’XTerm’))


# Match an xterm with a root shell (provided that the shell
# prompt sets a useful xterm title)

And(name(’XTerm’), re_title(r’\[root.*\]’))
```

8 Extension Modules

To actually get a useful window manager one must extend the core classes (see Chapter 4 [Core Classes], page 6) with various extension classes. Extension classes take the form of mixin classes, i.e. we just inherit it with the corresponding core class. They will add methods to the core class, and can usually be configured with class attributes.

For example, to create a client which highlights the window border when it is focused one could use this fragment:

```
class MyClient(wmanager.Client, border.BorderClient):
    pass
```

Because of the mixin technique, we need some ground rules for naming schemes, configuration and initialization. Finally some extension modules are described in detail. For full extension examples, look in the directory `examples` in the distribution tree.

8.1 Extension Coding Conventions

Extension classes will need their own member variables and methods, and to avoid classes overwriting the attributes of other classes the following naming scheme should be used:

- Each extension module should select a prefix, typically the same as the module name unless that it very unwieldy.
- All member variables and methods used and defined by an extension class should begin with the prefix.
- An exception: methods are allowed to begin with `get_` or `set_` followed by the prefix.

8.2 Extension Classes Initialization

Extension classes must be able to initialize themselves. To make this easy the core classes provide some special initializing functions for extension classes. Extension classes should only use these, they must not use the normal Python initialization funktion `__init__`.

When extending a core class by subclassing it together with a number of extension classes, the core class should be the first base class. The extension classes may have to be ordered among themselves too.

When an extended core class is initialized it will traverse the class inheritance tree. When an extension initialization function is found it is called without any arguments except for the object itself. As soon as an extension initialization function is found in a base class, its base classes will not be traversed.

`WindowManager` provides two extension initialization functions:

`__wm_screen_init__`
Called after the display is opened, but before any screens are added.

`__wm_init__`
Called after all the screens have been added, i.e. as the very last thing during the initialization of the window manager.

`Screen` also provides two extension initialization functions:

`__screen_client_init__`
Called after the root window has been fetched and the `EventDispatcher` has been created, but before any clients are added.

`__screen_init__`
Called after all the clients have been added, i.e. before the window manager adds the next screen.

`Client` provides only one extension initialization function:

`__client_init__`
Called after the client has finished all of its core initialization, i.e. just before the screen will add the next client.

There are also corresponding finalization methods:

`__wm_del__`
`__screen_del__`
`__client_del__`
These are called just before the object will finish itself off, similar to the `__del__` method of objects. (Actually, these methods are called from the `__del__` method of the object.)

8.3 Available Extension Modules

This is a description of the available extension modules, with information on how to use them and on the interface they provide to other extension modules.

8.3.1 `color` Extension Module

`color` provides a screen mixin for color handling:

Color [Screen Mixin]

`Color` handles color allocation. It maintains a cache of allocated colors to reduce PLWM's colormap footprint on displays with a low bit depth.

get_color (*color, default = None*) [Method on `Color`]

Returns the pixel value corresponding to *color*. *color* can be a string or tuple of (R, G, B) integers. If the color can't be allocated and *default* is provided, `get_color` tries to return that color instead. If that fails too, it raises a `ColorError` exception.

get_color_res (*res_name, res_class, default = None*) [Method on `Color`]

Return the pixel value for the color defined in the X resource identified by `res_name res_class`. `WindowManager.rdb_get` is used to lookup the resource, so the first components of the name and class should be omitted and they should start with '`.`'.

If `default` is provided, that name will be used if no matching X resource is found. If omitted, or if the color can't be allocated, `ColorError` is raised.

8.3.2 `font` Extension Module

`font` provides a window manager mixin for loading fonts:

Font [WindowManager Mixin]

Font provides two functions for loading fonts:

get_font (*fontname, default = None*) [Method on `Font`]

Returns the font object corresponding to *fontname*. If *fontname* doesn't match any font, attemt to return the font named *default* instead, if `default` is provided. If no font can be found, `FontError` is raised.

get_font_res (*res_name, res_class, default = None*) [Method on `Font`]

Return the font object corresponding to the X resource identified by `res_name` `res_class`. `WindowManager.rdb_get` is used to lookup the resource, so the first components of the name and class should be omitted and they should start with '`.`'.

If this resource isn't found or doesn't match any font, attempt to return the font named *default* instead, if *default* is provided.

If no font can be found, *FontError* is raised

8.3.3 `keys` Extension Module

`keys` provides two classes for handling key events: `KeyHandler` and its subclass `KeyGrabKeyboard`.

KeyHandler (*obj*) [Class]

Represents a key handler, and should only be used as a base class, never instantiated directly. Instantiate a class derived from KeyHandler to install its key handler. When instantiating, *obj* should be a `WindowManager`, `Screen`, or `Client` object.

If *obj* is a `WindowManager` object the key bindings defined will be active on all screens. If *obj* is a `Screen` object the key bindings will only be active when that screen is the current one. If *obj* is a `Client` object the key bindings will only be active when that client is focused.

propagate_keys [Instance Variable of KeyHandler]

This attribute controls whether this key handler will allow other key handlers to recieve events. If it is true, which is the default, key events will be passed to all currently installed key handlers. If it is false key events will only reach this key handler and other installed handlers will never see them.

timeout [Instance Variable of KeyHandler]

If this is set to a number, the key handler method `_timeout` will be called if no key has been pressed for `timeout` number of seconds. This is `None` by defalt, meaning that there are no timeout for this keyhandler.

_timeout (*event*) [Method on `KeyHandler`]

Called when the timeout is reached, if any. *event* is the `TimerEvent` causing the timeout. Key handlers using a timeout should override this method.

_cleanup ( ) [Method on `KeyHandler`]

Uninstall the key handler. This will remove all grabs held by the keyhandler, and remove its event handlers from the event dispatcher. Typically this is called from an overridden `_timeout`.

KeyGrabKeyboard (*obj*, *time*) [Class]

This `KeyHandler` subclass should be used when the application whishes to grab all key events, not only those corresponding to methods.

The *obj* argument is the same as for `KeyHandler`. *time* is the X time of the event which caused this key handler to be installed, typically the `time` attribute of the event object. It can also be the constant `X.CurrentTime`.

This class also changes the defaults for `propagate_keys` to false and `timeout` to 10 seconds, and provides a `_timeout` method which uninstalles the key handler.

A key handler is created by subclassing `KeyHandler` or `KeyGrabKeyboard`. All methods defined in the new key handler class represents represents key bindings. When a key event occures that match one of the methods, that method will be called with the event object as the only argument.

The name of the method encodes the key event the method is bound to. The syntax looks like this:

```
name :== keysym | modifiers '_' keysym

keysym :== <any keysym in Xlib.XK, without the XK_ prefix>

modifiers :== modifiers '_' modifier | modifier

modifier :== 'S' | 'C' | 'M' | 'M1' | 'M2' | 'M3' | 'M4' | 'M5' |
             'Any' | 'None' | 'R'
```

In other words, the method name should be a list of modifiers followed by the name of a keysym, all separated by underscores. The keysyms are found in the Python Xlib module `Xlib.XK`.

The modifiers have the following intepretation:

S	Shift
C	Control
M	Meta or Alt (interpreted as Mod1)
M1	Mod1
...	...
M5	Mod5
Any	Any modifier state, should not be combined with other modifiers
None	No modifiers, useful for binding to the key `9`, or other keysyms which are not valid method names by themselves
R	Bind to the key release event instead of the key press event

8.3.4 `focus` Extension Module

`focus` provides classes to track and control window focus changes. The core classes will generate events when focus changes. The order of the generated events is `ClientFocusOut`, `CurrentClientChange`, and `ClientFocusIn`. See Chapter 6 [Core Events], page 13.

PointToFocus [WindowManager Mixin]

This window manager mixin sets the current client to the one which currently contains the pointer. Most of the time, the current client also has focus. However if the current client don't use input, the previously focused client remains that.

SloppyFocus [WindowManager Mixin]

This is a subclass of `FocusHandler` which implements sloppy focus instead of point-to-focus. Sloppy focus means that a client will not loose focus when the pointer moves out to the root window, only when it moves to another client.

MoveFocus [WindowManager Mixin]

This mixin defines a method for moving focus between clients:

move_focus (*dir*) [Method on `FocusHandler`]

Move the focus to the next window in direction `dir`, which should be one of the constants `focus.MOVE_UP`, `focus.MOVE_DOWN`, `focus.MOVE_LEFT` or `focus.MOVE_RIGHT`.

Alas, this function is not very intelligent when choosing the next window, and it only works well when all windows are on a horizontal or vertical axis and focus is moved along that axis.

8.3.5 `border` Extension Module

This module provides a client mixin to change window border color depending on focus state. This module requires the `color` and `focus` modules to work properly.

BorderClient [Client Mixin]

Set a border on windows, and change its color depending on focus state. The colors used are set with the X resources `plwm.border.color/Plwm.Border.Color` and `plwm.border.focus.color/Plwm.Border.Focus.Color`. The defaults for these are `"`black`"` and `"`grey60`"`, respectively.

border_default_width [Instance Variable of BorderClient]

The border width in pixels. Default is 3.

no_border_clients [Instance Variable of BorderClient]

A client filter, used to select clients which should have no border. The default is `cfilter.false`, so all clients will have borders.

8.3.6 `outline` Extension Module

This module provides different ways of drawing an outline of windows. All outline classes are client mixins and have the same interface:

outline_show (*x = None, y = None, w = None, h = None, name = None*) [Method on `OutlineClient`]

Show an outline for this client's window. If an outline already is visible, it will be changed to reflect the arguments.

The arguments *x*, *y*, *w* and *h* gives the geometry of the outline. If any of these are not provided, the corresponding value from the current window geometry will be used.

If *name* is provided, that string will be displayed in the middle of the outline.

outline_hide ( ) [Method on `OutlineClient`]

Hide the outline, if it is visible.

Currently there are two outline classes:

XorOutlineClient [Client Mixin]

Draws the outline directly on the display, by xor-ing pixel values. The font used is set with the X resource `plwm.outline.font/Plwm.Outline.Font`. The default is `fixed`.

This is the most efficient outline method, but it has a few problems. If the windows under the outline changes, remains of the outline will still visible when it is hidden. The windows can be restored by e.g. iconifying and deiconifiying, switching to another view and back, or in an Emacs pressing `C-l`.

A bigger problem is that some combinations of depth, visual and colormap of the root window causes the xor of black to be black. This results in an invisible outline if you have a black background. This can be solved by changing the background colour of the root, or using some other outline method.

WindowOutlineClient [Client Mixin]

This "draws" the outline by creating a set of thin windows, simulating drawing lines on the screen. Any name is displayed by drawing it in a centered window, using the font specified as above.

This is less efficient than an xor outline, since eight or nine windows have to be moved and resized if the outline is changed. However, it does not have any of the problems listed for `XorOutlineClient`.

The colours used is currently hardcoded to black and white.

8.3.7 `moveresize` Extension Module

This module provides functionality for moving and resizing windows. The core functionality is implemented by the abstract base class `MoveResize`. It is subclassed by the two classes `MoveResizeOpaque` and `MoveResizeOutline` which resizes windows by changing the window size, and by drawing an outline of the new size, respectivelly. The latter requires the `outline` module. See the code for details on these classes.

Most resizing will be done via key handlers, so a template key handler class is provides that simplifies writing your own moving and resizing keyhandler for the currently focused client:

MoveResizeKeys (*from_keyhandler, event*) [Class]

`MoveResizeKeys` contains methods for the various move and resize operations. Is should be subclassed, and in the subclass key binding method names should be assigned to the general methods.

There are 24 general methods:

_move_X	Move the client in direction X
_enlarge_X	Enlarge the client in direction X
_shrink_X	Shrink the client from direction X

The direction is one of eight combinations of the four cardinal points: e, ne, n, nw, w, sw, s and se.

Additionally theres two methods for finishing the moveresize:

_moveresize_end	Finish, actually moving and resizing the client
_moveresize_abort	Abort, leaving client with its old geometry

By default outline moveresizing is used with the `MoveResizeOutline` class. This can be changed by redefining the attribute `_moveresize_class` to any subclass of `MoveResize`.

A small `MoveResizeKeys` subclass example:

```
class MyMRKeys(MoveResizeKeys):
    _moveresize_class = MoveResizeOpaque

    KP_Left = MoveResizeKeys._move_w
    KP_Right = MoveResizeKeys._move_e
    KP_Up = MoveResizeKeys._move_n
    KP_Down = MoveResizeKeys._move_s

    KP_Begin = MoveResizeKeys._moveresize_end
    Escape = MoveResizeKeys._moveresize_abort
```

This would be invoked like this in a keyhandler event method in your basic keyhandler:

```
def KP_Begin(self, evt):
    MyMRKeys(self, evt)
```

`MoveResize` generates events during operation. The `type` attribute for all these are the event class, and the `client` attribute is the affected client.

`MoveResizeStart` is generated when the moveresize is started, `MoveResizeEnd` when it ends and the window geometry is changed, and `MoveResizeEnd` when it is aborted and the window geometry is left unchanged.

`MoveResizeDo` is generated for each change in window geometry during moveresize. It has four attributes denoting the current geometry: `x`, `y`, `width` and `height`.

8.3.8 `cycle` Extension Module

`cycle` provides classes for cycling among windows to select one of them to be activated. This is performed by an abstract base class:

Cycle (*screen, client_filter*) [Class]

Cycle among the windows on *screen* matching *client_filter*.

next ( ) [Method on `Cycle`]

Cycle to the next window.

previous ( ) [Method on `Cycle`]

Cycle to the previous window.

end ( ) [Method on `Cycle`]

Finish and activating the selected window.

abort ( ) [Method on `Cycle`]

Abort, not activating the selected window.

This is implemented by two subclasses: `CycleActive` which cycles among windows by activating them in turn, and `CycleOutline` which cycle among windows by drawing an outline of the currently selected window. The latter requires the `outline` extension.

To simplify writing a key handler for cycling, a template key handler is provided:

CycleKeys (*keyhandler, event*) [Class]

Cycle among the windows on the current screen matching the client filter specified by the attribute `_cycle_filter`. This is `cfilter.true` by default, cycling among all windows. The cycle method is specified by the attribute `_cycle_class`, which by default is `CycleOutline`.

CycleKeys defines a number of event handler methods:

_cycle_next	Cycle to the next client
_cycle_previous	Cycle to the previous client
_cycle_end	Finish, selecting the current client
_cycle_abort	Abort, reverting to the previous state (if possible)

A small `CycleKeys` subclass example:

```
class MyCycleKeys(CycleKeys):
    _cycle_class = CycleActivate
    _cycle_filter = cfilter.Not(cfilter.iconified)

    Tab = CycleKeys._cycle_next
    C_Tab = CycleKeys._cycle_next
    S_Tab = CycleKeys._cycle_previous
    S_C_Tab = CycleKeys._cycle_previous

    Return = CycleKeys._cycle_end
    Escape = CycleKeys._cycle_abort
```

To activate your cycle keys, write a keyhandler event method like this in your basic keyhandler:

```
    def C_Tab(self, evt):
        MyCycleKeys(self, evt)
```

8.3.9 `views` Extension Module

Views are PLWM's "workspaces". Many window manager have the concept of workspaces, or virtual screens. They give the illusion of having several screens, although only one of them can be displayed at a given time on the physical screen. This is done by iconifying the windows not visible on a workspace when that workspace is displayed.

Views does this too, and more. A view can be seen as a projection of the avialable windows onto the screen in a certain configuration. Views not only remembers which windows are visible on them, but also their geometry and stacking order. This means that the same window can appear on several views in a different place, even with a different size, on each view. Views also remembers the pointer position when swapping to another view, and restores it when the view is activated again. All information about the view configuration is stored when PLWM exits, so it can be restored after a restart.

Additionally, you can create views dynamically when you need them, and when they are no longer needed they will be destroyed. A view is considered to be unneeded if it is empty when you switch to another view.

All this is handled by the screen mixin `ViewHandler`:

ViewHandler [Screen Mixin]

view_always_visible_clients [Instance Variable of ViewHandler]

A client filter matching the client windows that should be visible on all views, irrespective of view configuration. When determining if a view is empty so it can be deleted, these windows will be ignored. The default value is `cfilter.false`.

view_new (*copyconf = 0*) [Method on `ViewHandler`]

Create a new view and switch to it. If *copyconf* is true, the window configuration of the current view will be copied, otherwise the new view will be empty.

view_next ( ) [Method on `ViewHandler`]

Switch to the next view.

view_prev ( ) [Method on `ViewHandler`]

Switch to the previous view.

view_goto (*index, noexc = 0*) [Method on `ViewHandler`]

Switch to view number *index*, counting from 0. If *noexc* is false `IndexError` will be raised if *index* is out or range. If *noexc* is true, quitely return.

view_find_with_client (*clients*) [Method on `ViewHandler`]

Switch to the next view where there is a visible client matching the client filter *clients*. Beeps if there none.

view_tag (*tag*) [Method on `ViewHandler`]

Set a tag on the current view. *tag* can be any string.

view_find_tag (*tag*) [Method on `ViewHandler`]

Switch to the next view with tag *tag*. Beeps if there is none.

If there is a modewindow, one can use the mixin `XMW_ViewHandler` instead of `ViewHandler` to get information on the current view number and tags in the modewindow.

8.3.10 `menu` Extension Module

The `menu` module provides a screen mixin to display a menu of options for the user to select from, as well as a pair of keyboard handler templates for the menu. There is only one menu for each screen, but the available options may be changed each time it is displayed. To the user, there appear to be many menus, but only one may be displayed at a time.

screenMenu [Screen Mixin]

Provides the menu window for each screen. The look of the window is controlled by the following class variables:

Variable	Default	Description
menu_fontname	9x15bold	Font for menu options.
menu_foreground	black	Foreground color for the menu window.
menu_background	white	Background color for the menu window.
menu_borderwidth	3	Border for the men window.
menu_handler	MenuKeyHandler	Keyboard handler for menus.

menu_make (*labels*, *align = 'center'*) [Method on `screenMenu`]

Creates a menu window from *labels*, which must be a sequence of strings. The strings will be aligned in the window according to the value of *align*, which may be `'left'`, `'right'` or `'center'`. The *width* and *height* of the resulting window are returned as a tuple for use in calculating the menu placement.

menu_run (*x*, *y*, *action*) [Method on `screenMenu`]

x and *y* are the coordinates the menu should be placed at. *action* is a callable argument that will be invoked with the string used for the label the user selected. If the user aborts the menu, action will not be invoked.

A simple example of a menu with dictionary of functions might be:

```
class MyFunctionMenu:

    def __init__(self, screen, dict):
        self.dict = dict
        labels = dict.keys()
labels.sort()
width, height = screen.menu_make(labels)
# Center the menu
screen.menu_run((screen.root_width - width) / 2,
                        (screen.root_height - height) / 2,
self)

    def __call__(self, choice):
        self.dict[choice]()
```

Making selections and aborting the menu are done via key handlers See Section 8.3.3 [keys], page 18, and two template key handlers are provided for menu selections:

MenuKeyHandler [Class]

`MenuKeyHandler` provides the methods `_up`, `_down`, `_do` and `_abort`. These move the current selection, pass the current selection to the *action* object passed to `menu_run`, and abort the menu taking no action. A binding with Emacs keys might look like:

```
class MyMenuKeys(MenuKeyHandler):
      C_p = MenuKeyHandler._up
      C_n = MenuKeyHandler._down
      Return = MenuKeyHandler._do
      C_g = MenuKeyHandler._abort
```

MenuCharHandler [Class]

`MenuCharHandler` adds the `_goto` method, which moves the current selection to the first label that starts with the a character greater than or equal to the typed key. It then binds the keys `a` to `z` and `0` to `9` to _goto. This lets the user select labels by their first character if `MenuCharHandler` is used instead of `MenuKeyHandler`.

To have menus on your screen use your menu keys, you would add the `screenMenu` mixin and set the `menu_handler` class variable:

```
class MyScreen(Screen, screenMenu):
      menu_handler = MyMenuKeys
```

8.3.11 `panes` Extension Module

The `panes` mixins provide an alternative method of managing windows. Rather than wrapping each window in a frame which is manipulated to manipulate the window, windows are placed in "panes", and the only thing the user can do to windows in a pane is circulate through them. When a window is placed in a pane, it will be resized to the largest size that it can handle which will fit in that pane. However, panes can be split into two parts at whatever fraction of the full pane the user chooses, so that panes can be created with nearly arbitrary geometry. Panes do not overlap, and every pixel on the screen is in a pane.

See ‘`examples/plpwm.py`’ for an example of using panes to build a window manager.

Pane [Class]

add_window (*client*) [Method on `Pane`]

Adds the clients window to the current pane. It will become the top window in the pane. If the current top window in the pane has focus, the new top window will get focus.

iconify_window ( ) [Method on `Pane`]

Iconifies the panes active window.

force_window ( ) [Method on `Pane`]

Resize the window again. This actually resizes the window down then back up, and is useful if the application doesn’t realize how big the window really is. This is most often seen in programs started in an xterm by the xterm command.

next_window ( ) [Method on `Pane`]

Make the next window associated with this pane the top window.

prev_window ( ) [Method on `Pane`]

Make the previous window associated with this pane the top window. When a window is added to a pane, the window that was the top window becomes the previous window.

horizontal_split (*fraction = .5*) [Method on `Pane`]

Split the current pane into two halves horizontally. The new pane will get *fraction* of the current panes height at the bottom of the current pane, and will become the active pane.

vertical_split (*fraction = .5*) [Method on `Pane`]

Split the current pane into two halves vertically. The new pane will get *fraction* of the current panes width at the right of the current pane, and will become the active pane.

maximize ( ) [Method on `Pane`]

Make the current pane occupy the entire screen, removing all other panes.

panelfilter (*pane*) [Filter]

True if the client is in the given pane.

panesManager [WindowManager Mixin]

The `panesManager` mixin adds panes and pane manipulation to the window manager.

panes_list [Instance Variable of panesManager]

The list of panes managed by this windowmanager.

panes_current [Instance Variable of panesManager]

The index of the pane containing the currently active window, also known as the active pane.

panes_window_gravity [Instance Variable of panesManager]

The gravity to be used for normal windows in this pane.

panes_maxsize_gravity [Instance Variable of panesManager]

The gravity to be used for windows with maxsize hints in this pane.

panes_transient_gravity [Instance Variable of panesManager]

The gravity to be used for transient windows in this pane.

panes_goto (*index*) [Method on `panesManager`]

Make the pane at *index* in `panes_list` the active pane.

panes_activate (*pane*) [Method on `panesManager`]

Make *pane* the active pane.

panes_next ( ) [Method on `panesManager`]

Make the next pane in the list the active pane. If the last pane is the active pane, make pane 0 the active pane.

panes_prev ( ) [Method on `panesManager`]

Make the previous pane in the list the active pane. If pane 0 was active, make the last pane in the list active.

panes_number (*number*) [Method on `panesManager`]

Rearrange `panes_list` so that the active pane is pane *number*. This is done by exchanging the list positions of pane *number* and pane `panes_current`.

panes_save ( ) [Method on `panesManager`]

Save the state of all clients by building a dictionary of which pane they are associated with.

panes_restore ( ) [Method on `panesManager`]

Put all clients back in the pane they were in when panes_save was last invoked, if possible. If the pane doesn't exist or is on a different screen from the window, the restore isn't possible.

panesScreen [Screen Mixin]

This mixin causes the first pane to be created on each screen being managed. It has no variables or methods useful to the user, but you must mix it into your screen class if you want to use panes.

panesClient [Client Mixin]

This mixin passes client events to the pane that the client's window is associated with. It has no variables or methods useful to the user, but you must mix it into your client class if you want to use panes.

8.3.12 `modewindow` Extension Module

The `modewindow` module provides a screen mixin to display a window containing general window manager information, and a class representing this information. The name of the module derives from the mode-line in Emacs, which has a similar function.

ModeWindowScreen [Screen Mixin]

Displays a mode window on the screen. The look of the window is controlled with the following X resources:

plwm.modewindow.foreground/Plwm.ModeWindow.Foreground
plwm.modewindow.background/Plwm.ModeWindow.Background

These set the colors to be used by the modewindow. Defaults are black foreground and white background.

plwm.modewindow.font/Plwm.ModeWindow.Font

The font to use in the modewindow. Default is fixed.

modewindow_pos [Instance Variable of ModeWindowScreen]

Controls the position of the mode window. Can either be `modewindow.TOP` or `modewindow.BOTTOM`.

modewindow_add_message (*message*) [Method on `ModeWindowScreen`]

Add *message*, which must be a `Message` object, to this mode window. A single message object can be added to several mode windows.

modewindow_remove_message (*message*) [Method on `ModeWindowScreen`]

Remove *message* from this modewindow.

Message (*position, justification = modewindow.CENTER, nice = 0, text = None*) [Class]

Represents a single message to be displayed in one or more mode windows.

position is the horizontal position for this message in the modewindow, and should be a float in the range [0.0, 1.0]. The message text is drawn at this point according to *justification*, which should be one of the values `modewindow.LEFT`, `modewindow.CENTER` or `modewindow.RIGHT`. *nice* is currently not used, but is meant to be used to avoid message overlaps by shuffling less important messages around. Finally, *text* is the initial text of this message, where `None` means an empty message.

set_text (*text*) [Method on `Message`]

Change the text of this message to *text*. All affected mode windows will be redrawn, if necessary.

8.3.13 `modestatus` Extension Module

`modestatus` is a layer on top of `modewindow`, providing a way to display the current status of the window manager, e.g. the focused window, the geometry of a window during resize, and more.

ModeStatus [Screen Mixin]

Add a status message to the center of this screen's mode window. The status message is really a stack of different messages, where the top-most message is currently displayed.

modestatus_set_default (*text*) [Method on `ModeStatus`]

Set the default text to be displayed when there is no special status to *text*.

modestatus_new (*text = ''*) [Method on `ModeStatus`]

Push a new message on to the status message stack. A `ModeText` object will be returned, which will have *text* as the initial message.

ModeText [Class]

This class should never be instantiated directly, only through `ModeStatus.modestatus_new`.

set (*text*) [Method on `ModeText`]

Set the text of this status message to *text*. If this is the top-most message, the new text will be displayed.

pop ( ) [Method on `ModeText`]

Remove this message from the message stack. If this was the top-most message, the previous message will be displayed instead.

This module also provides some mixins that use the mode status functionality:

ModeFocusedTitle [Client Mixin]

Display the currently focused client's title as the default message.

ModeMoveResize [Screen Mixin]

When moving and resizing, display the title of the displayed window and the current geometry. The format of the message is controlled by an X resource with name `plwm.moveResize.modeFormat` and class `Plwm.MoveResize.ModeFormat`. It should be a Python format string, where `%(title)s` will be replaced with the client title, and `%(geometry)s` with the current geometry. The default format is `%(title)s [%(geometry)s]`.

8.3.14 `mw_clock` Extension Module

ModeWindowClock [WindowManager Mixin]

This mixin displays the current time in all mode windows. It is updated once a minute. The format is a `time.strftime` format, by default `%H:%M`. It can be changed with an X resource with name `plwm.modewindow.clock.format` and class `Plwm.ModeWindow.Clock.Format`.

mw_clock_position [Instance Variable of ModeWindowClock]

The position of the time message in the mode window, default 1.0.

mw_clock_justification [Instance Variable of ModeWindowClock]

The justification of the time message, default is `modewindow.RIGHT`.

8.3.15 `mw_biff` Extension Module

This module provides two different mail notifications mixins, which both use the mode windows and beeping for notification. They assume that new mail is stored in `$MAIL`, and removed from it when read. This works well with the behaviour of Gnus with the nnmail backend.

When `$MAIL` is empty or non-existent, no message is displayed. When new mail arrives the message '`New mail`' is displayed, and the speaker beeps. If `$MAIL` is accessed without being emptied, the message is changed to '`Mail`'. When `$MAIL` is emptied, the message is removed again.

The messages can be changed with X resources. The X resource with name `plwm.modewindow.newMail.text` and class `Plwm.ModeWindow.NewMail.Text` controls the new mail message, and the resource with name `plwm.modewindow.Mail.text` and class `Plwm.ModeWindow.Mail.Text` controls the mail exists message.

Changing the mail exists message to '' could be useful if one uses a mail client that leaves read mail in `$MAIL`.

ModeWindowBiff [WindowManager Mixin]

Displays a mail notification message in all mode windows.

mw_biff_position [Instance Variable of ModeWindowBiff]

The position of the mail message in the mode window, default 0.0.

mw_biff_justification [Instance Variable of ModeWindowBiff]

The justification of the mail message, default is `modewindow.LEFT`.

If the mailspool is mounted over NFS, it might be unadvisable to access it from the window manager. If the NFS server should freeze, the entire window manager would be unusable until the server recovers. Therefore a threaded biff mixin is provided:

ThreadedModeWindowBiff [WindowManager Mixin]

This subclasses `ModeWindowBiff`, with the change that access to `$MAIL` is done in a separate thread. As PLWM certainly isn't thread-safe, all interaction with the rest of the window manager modules is done in the main thread. Communication between the mailspool access thread and the main thread is done without locks or semaphores. The main thread polls at regular intervals whether the access thread has finished yet, and if so, updates the mode window. This uses the property of Python threading that only one thread at a time may access Python objects. If this would ever change, this class might break.

8.3.16 `mw_apm` Extension Module

ModeWindowAPM [WindowManager Mixin]

This mixin displays the battery status in all mode windows. There is a generic interface for fetching the status, making it easy to port this module to different APM systems. Currently, the only supported system is the special file '`/proc/apm`' of Linux systems.

mw_apm_position [Instance Variable of ModeWindowClock]

The position of the battery status in the mode window, default 0.2.

mw_apm_justification [Instance Variable of ModeWindowClock]

The justification of the battery status, default is `modewindow.RIGHT`.

8.3.17 `input` Extensions Module

`Input` provides tools to let the window manager read a line of input from the user and act on it. It provides a subclass of `KeyGrabKeyboard` (see Section 8.3.3 [keys], page 18) for configuration, and two classes to read input.

InputKeyHandler (**handler, displayer**) [Class]

Creates a key handler class for editing input. The *handler* is any object acceptable to `KeyGrabKeyboard`. *displyer*`.show(`*`left, right`*`)` is called to display the two strings with a curser between them. *displayer*`.do(`*`text`*`)` is called when the user is through editing the input. *displayer*`.abort()` is called if the user aborts the operation.

`InputKeyHandler` provides the following methods that may be bound to keystrokes like any other keyhander. See Section 8.3.3 [keys], page 18.

_insert (*event*) [Method on `InputKeyHandler`]

The key pressed to generate `event` is inserted into the buffer at the cursor. All the characters of the latin1 character set are bound to this by default.

_forw (*event*) [Method on `InputKeyHandler`]

Move the cursor forward one character in the edited text.

_back (*event*) [Method on `InputKeyHandler`]

Move the cursor backward one character in the edited text.

_delforw (*event*) [Method on `InputKeyHandler`]

Delete the character in front of the cursor in the edited text.

_back (*event*) [Method on `InputKeyHandler`]

Delete the character behind of the cursor in the edited text.

_end (*event*) [Method on `InputKeyHandler`]

Move the cursor to the end of the edited text.

_begin (*event*) [Method on `InputKeyHandler`]

Move the cursor to the beginning of the edited text.

_back (*event*) [Method on `InputKeyHandler`]

Delete all the characters from the cursor to the end of the edited text.

_paste (*event*) [Method on `InputKeyHandler`]

If text is selected, it will be inserted into the edited text at the cursor, as if typed by the user. For this to work, the `handler` must be an instance of `wmanager.Window`. If that is not the case, or no text is currently selected, this does nothing.

_done (*event*) [Method on `InputKeyHandler`]

Finishes the action, and calls `displayer.do` passing it the edited text.

_abort (*event*) [Method on `InputKeyHander`]

Aborts the input operation, callgin `displayer.abort()`.

inputWindow (prompt, screen, length = 30) [Class]

`inputWindow` is a class that creates a window on *screen* and uses an `InputKeyHandler` to read input from the user. The user is prompted with *prompt*. Space is left for *length* extra characterfs to display in the window, but it will scroll to keep the cursor always in view.

fontname [Instance Variable of inputWindow]
foreground [Instance Variable of inputWindow]
background [Instance Variable of inputWindow]
borderwidth [Instance Variable of inputWindow]

The attributes of the window used to read the input. The defaults are 9x15 for the *fontname*, a black *foreground* on a white *background*, and a *borderwidth* of 3.

height [Instance Variable of inputWindow]
width [Instance Variable of inputWindow]

The *height* and *width* of the window will be available as attributes after the `inputWindow` is instantiated.

read (*action, handlertype, x = 0, y = 0*) [Method on `inputWindow`]

The read method of the inputWindow is called to read text from user and act on it. *handlertype* should be a subclass of `InputKeyHandler` with appropriate bindings for editing the text. *action* will be invoked with the edited text as it's sole argument when *handlertypes*'s `_done` action is invoked.

modeInput (**prompt, screen**) [Class]

An `modeInput` object uses the `modewindow` on *screen* to read input from the user. See Section 8.3.12 [modewindow], page 28. It has an `read` method that takes the same arguments as the `read` method of `inputWindow`, except that x and y are ignored.

8.3.18 `inspect` Extension Module

`Inspect` allows a special client to remotly connect to the running window manager. The client can then execute Python statements in the context of the window manager. This can be used to inspect the internal state of the window manager, change it, or whatever you might come up with.

The server side is implemented by a window manager mixin. For documentation on the client program, see Section 9.2 [inspect_plwm], page 34.

InspectServer [Window Manager Mixin]

The inspect server can be enabled or disabled. When it is enabled, the message `[Inspect]` is showed in the modewindow. When inspect clients connect it will change to also show the number of connected clients.

[Instance Variable of inspect_enabled_at_start]

Whether the inspect server should be enabled at startup. Default value is false.

inspect_enable ( ) [Method on `InspectServer`]

Enable the inspect server.

inspect_disable (*force = 0*) [Method on `InspectServer`]

Disable the inspect server. If *force* is false disabling will fail if clients are connected, which will be indicated by a beep. If *force* is true the clients will be disconnected and the inspect server disabled.

inspect_toggle (*force = 0*) [Method on `InspectServer`]

Toggle the inspect server on or off. *force* is only used for disabling, and has the same meaning as for `inspect_disable`.

The inspect server is as secure as your X display. The inspect server announces the TCP port it listens on in a property on the root window. In the same property it also stores a random, 31-bit cookie. To be able to connect to the inspect server the client must fetch this property to find the port to connect to, and the cookie to send as authorization. Therefore only those with access to your X display, thanks to xhost or xauth, can connect to the inspect server.

9 Utilities

The following small X programs can be seen as a first step towards making PLWM a full pointless desktop to rival GNOME and KDE. Well, maybe not.

9.1 wmm

wmm, the Window Manager Manager, is a utility to simplify testing your freshly hacked window manager. It opens a small window containing one or more buttons. When a button is clicked, a command is executed.

The window manager managing feature is that if wmm is iconified by the running window manager it will be mapped when the window manager exits or crashes. When wmm detects that it has been mapped it will raise itself to the top of all the other windows. This ensures that it will be possible to click on one of its buttons to start another window manager, and thus be able to fix the bug which lurked in your window manager.

wmm is configured with `~/.wmmrc`. For each non-blank line, not starting with a #, wmm will create a button. Each line should consist of two tab-separated fields, the first is the button label and the second is the command to run when the button is clicked (it should probably end with an `&` so WMM isn't locked). If the second field is missing, wmm will instead quit when the button is clicked.

9.2 inspect_plwm

This utility is used to connect to the inspect server of a running window manager. The client must be able to connect to the display that the window manager is running on to be able to connect to the inspect server. The display is fetched from the `$DISPLAY` variable or the command line option `-display`.

When successfully connected, a welcome message is displayed by the window manager and a common Python prompt is shown. Python expressions and statements can now be entered. They will be evaluated in the window manager, and the results or tracebacks will be printed.

The code will be evalated in an environment containing all builtin functions and the variable `wm`, which points to the WindowManager object representing the window manager.

An example session (long lines have been wrapped):

```
[petli@sid petli]$ inspect_plwm
Welcome to PLWM at :0
>>> wm
<__main__.PLWM instance at 82148a8>
>>> wm.default_screen.clients.values()
[<__main__.MyClient instance at 822ec28>,
 <__main__.MyClient instance at 8215ce8>,
 <__main__.MyClient instance at 8217370>,
 <__main__.MyClient instance at 822bfc0>]
>>> import sys
>>> map(lambda c: sys.stdout.write(c.get_title() + '\n'),
         wm.default_screen.clients.values())
xterm
xterm
WMManager
emacs@sid.cendio.se
[None, None, None, None]
>>>
```

Note that modules can be imported, and that sys.stdout and sys.stderr will output in the terminal window inspect_plwm is running in, even though the expressions are evaluated inside the window manager.

Multi-line statements can also be written with a small kludge: The lines must start with exactly one space, and one signals that the suite is finished by entering an empty line (without any space at all). Example:

```
>>>  for c in wm.default_screen.clients.values():
...    print c.get_title(), c.geometry()
...
xterm (516, 30, 502, 732, 3)
xterm (0, 30, 508, 732, 3)
WMManager (100, 0, 63, 71, 3)
emacs@sid.cendio.se (192, 18, 632, 744, 3)
>>>
```

10 ToDo

Known bugs:

- None, right now. But there are probably still memory leaks, subversive behaviour and smelly code here and there.

Fairly simple improvements:

- X resources: get rid of them. Or at least use Client/Screen/WindowManager attributes in the first place, falling back on resources for backward compitability.
- modewin: Teach it to avoid overlapping texts. Currently it will only write overlapping texts on top on each other, but it'd be nice if it was intelligent enough to shuffle the texts around to avoid it.
- Being able to dynamically turn on and off debugging in a running PLWM. Most of the framework is there, it only needs some key bindings.

More advanced fixes and features:

- Improve inspection to allow pdb debugging.
- Provide a class which can represent the layout of the windows in such a way that one can easily ask for things like "the window to the left of this window", "the first window edge we run into moving this window upwards" or "the visible parts of this window".
- Real frames around the client windows. This requires reparenting the windows and thus quite a number of modifications to wmanager.Client.
- Maybe some people would like mouse support?
- Other focus methods, such as click-to-focus.

Real out-of-this-time features:

- Support for controlling specific clients from the keyboard. Ex: pressing tab in a Netscape window would move the pointer to the next hyperlink in the document. Update: Okay, Netscape 6.1 actually have tabbing between links. There are probably still uses for this idea, and the modification to `keys` in 2.3 was done to make this possible.
- Rewrite the whole thing as threaded collection of agents, or something. PLWM is beginning to feel quite bulky.

Misc stuff:

- Replace autoconf script with Distutils script. Or maybe a combination of both.
- Change the window script idea around, so that a small script is installed as `plwm` which then simply sources '`~/.plwm.py`'.

11 Credits

PLWM was born late one night in the spring of 1999 when Peter Liljenberg and Morgan Eklf, as our habit is, enjoyed music and conversation. After some general window manager discussions and consensus on the beauty of using Python instead of yet another configuration file language, the name `"the Pointless Window Manager"` popped up. It was so good that we just had to implement it.

When it came to hacking, Peter was more inclined to let the work (or life) suffer and as a result have written all of the code.

Henrik Rindlw has helped with ironing out bugs triggered in multiheaded environments. By being the first other active user of PLWM he has also found quite a number of more normal bugs.

Our now former employer Cendio Systems (`http://www.cendio.se/`) deserves a paragraph here. The employee contracts explicitly mentioned that we were allowed to develop non-work-related GPL'd programs using the company's computers in our spare time, and keep the copyright. Nice. Now I can admit that quite a lot of work time also went into the development...

Mike Meyer wrote the very interesting extension modules `panes` and `menu`, and the corresponding example window manager `plpwm`.

12 Contact Info

Bug reports, feature requests, new modules, bug fixes, etc, should go to Peter Liljenberg <petli@ctrl-c.liu.se>.

New versions of PLWM will be announced on the PLWM website (`http://plwm.sourceforge.net/`) and on Freshmeat (`http://www.freshmeat.net/`).

PLWM is a SourceForge project. Mailinglists, bug tracking and public CVS access can be found at the project page (`http://sourceforge.net/project/plwm/`).