

# **Floating-Window-Manager wmwm um Tiling-Funktion erweitern**

Andreas Müller

27. September 2016

Umschüler zum Fachinformatiker/Anwendungsentwicklung

Ausbildungsbetrieb: media project academy GmbH  
Glashütter Straße 101  
01277 Dresden

Betreuer: Wilfried Bergler

Projektzeitraum: 05.09.2016 - 23.09.2016

# Inhaltsverzeichnis

<b>1. Einführung</b>	<b>1</b>
1.1. Floating-Window-Manager . . . . .	2
1.2. Tiling-Window-Manager . . . . .	2
1.2.1. Constraints-basiertes Tiling . . . . .	3
1.2.2. Linear organisiertes Tiling . . . . .	3
1.2.3. Baum-basierende Window-Manager . . . . .	4
<b>2. Entscheidung</b>	<b>4</b>
<b>3. Projektumgebung</b>	<b>5</b>
<b>4. Projektverlauf</b>	<b>5</b>
4.1. Vorarbeiten . . . . .	5
4.2. Baumstruktur . . . . .	5
4.3. Integration der Tiling-Funktionalität . . . . .	7
<b>5. Literaturverzeichnis</b>	<b>9</b>
<b>6. Glossar</b>	<b>10</b>
<b>A. Anlagen</b>	<b>I</b>
A.1. tree.h . . . . .	I
A.2. window_tree.h . . . . .	II
<b>B. Kundendokumentation</b>	<b>IV</b>
B.1. Einleitung . . . . .	IV
B.2. Bedienung . . . . .	V
B.3. Konfiguration . . . . .	V

## Aufgabenstellung

Der Window-Manager *wmwm* für das X Window System (X) unter Linux soll um eine Tiling-Funktionalität erweitert werden. So sollen Fenster wahlweise unter Ausnutzung des vollen Bildschirmes aufgeteilt werden oder alternativ frei bewegbar sein wie zuvor.

### 1. Einführung

Ein X Window-Manager ist ein X-Client-Programm zum Verwalten und Manipulieren von Fenstern unter X[1, S.10,12], er kann dazu dienen ein Fensterlayout durchzusetzen und eine Benutzeroberfläche zum Bedienen mehrerer Fenster und Monitore bereit zu stellen[2, S.66].

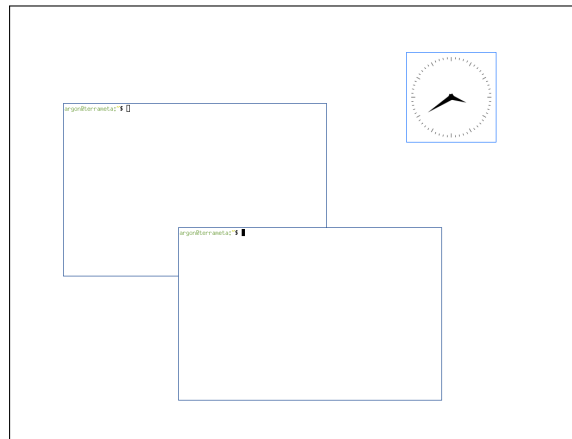


Abbildung 1: *Ursprünglicher wmwm mit Floating-Fenstern*

#### **wmwm**

Der vorgegebene X Window-Manager ist ein Fork von *mcwm*[3]. Dieser 2010 von Michael Cardell Widerkrantz geschriebene Window-Manager ist minimalistisch angelegt, in C für X11 unter Linux geschrieben und nutzt die xcb-Bibliothek um mit dem X11-Server zu kommunizieren. Es gibt keine Fensterdekoration außer einer 1-Pixel breiten Umrandung des jeweiligen Fensters. Er bietet mehrere Workspaces auf denen Fenster hinterlegt werden können. Diese Fenster können beliebig verschoben und in der Größe verändert werden. Mehrere Monitore werden unterstützt. Widerkrantz orientierte sich hierbei stark an *evilwm*[4], dem vorher verwendeten Window-Manager und *tinywm*[5, 6]. *mcwm* ist unter der ISC-Lizenz als Open Source veröffentlicht.

Im Jahr 2012 begann ich selbst den mcwm zu benutzen. Im Zuge der Bereinigung von Fehlern im Programmcode (Bugs) und Anpassungen begann zu Codebasis zunehmend vom Original zu divergieren. Im März 2015 importierte ich den bis dahin veränderten Quellcode in ein lokales git-Repository und benannte den Fork am 27.02.2016 in *wmwm* um. Der Quelltext ist weiterhin Open Source aber zum Zeitpunkt des Projektes nicht veröffentlicht. Im Rahmen meiner Ausbildung zum Fachinformatiker/Anwendungsentwicklung lernte ich Tiling-Window-Manager kennen und wünschte diese Funktionalität auch im *wmwm* zu implementieren. Die Projektarbeit ermöglichte mir dies durchzuführen.

Einige Arten von Window-Managern sollen nachfolgend erläutert werden.

### 1.1. Floating-Window-Manager

Ein Floating-Window-Manager (auch Stacking Window-Manager) stellt es dem Benutzer frei wo und wie die Fenster platziert werden, Fenster können überlappen, gestapelt sein, und eine beliebige Größe haben (siehe Abbildung 1). Dies wird auch als *Desktop Metapher* bezeichnet, da Fenster hier wie Dokumente auf einem Schreibtisch übereinander liegen können[7, S.35-36][2, S.67].

Hier offenbart sich auch ein Problem, durch das Überdecken können Informationen teilweise oder ganz verborgen sein und man kann leicht die Übersicht über die geöffneten Fenster verlieren. Auch ist die Aufteilung des Bildschirmes zur optimalen Ausnutzung der Bildfläche dem Nutzer überlassen und kostet Aufwand und Zeit zum Organisieren[7, S.35][8, S.36]

### 1.2. Tiling-Window-Manager

Ein Tiling Window-Manager (vgl. Abbildung 2) garantiert hingegen grundsätzlich, dass Fenster sich nicht überlappen können. Je nach Implementation kümmert er sich auch um die Aufteilung der verfügbaren Bildschirmfläche. Moderne Tiling Window-Manager beinhalten auch einen Floating-Window-Modus für Fenster die voraussichtlich beim Integrieren in die Tiling Anordnung störend wären und den Arbeitsfluss so behindern könnten[8, S.37]. Darunter fallen z.B. Dialogfenster oder Fenster mit Warnungen die nach einer kurzen Interaktion mit dem Nutzer wieder geschlossen werden. Sie werden *Transient Windows* genannt (auch [9],[10, Sec.4.1.2.6]).

Es gibt unterschiedliche Arten der automatischen Aufteilung von Fenstern. Mischformen sind nicht ausgeschlossen.

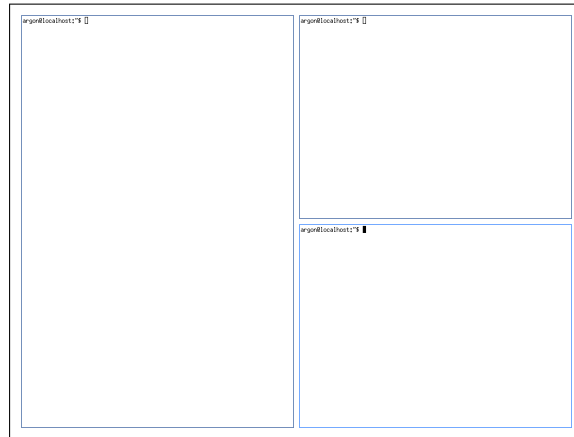


Abbildung 2: *wwwm* mit 3 Fenstern im *Tiling-Modus*

### 1.2.1. Constraints-basiertes Tiling

Ein Constraints-basierender Tiling Window-Manager (auch *Freies Tiling*) versucht anhand gegebener Zwänge und am freien Bereich[11, S.2], die Fenster möglichst optimal im verfügbaren Raum zu verteilen. Diese Zwänge leiten sich von Informationen des Fensters und Benutzereingaben her. So kann der Nutzer z.B. festlegen, dass ein Fenster immer mindestens eine bestimmte Größe oder Position haben soll. Es gibt keine inhärente Ordnung der Fenster (nicht-hierarchische Organisation[7, S.37]).

### 1.2.2. Linear organisiertes Tiling

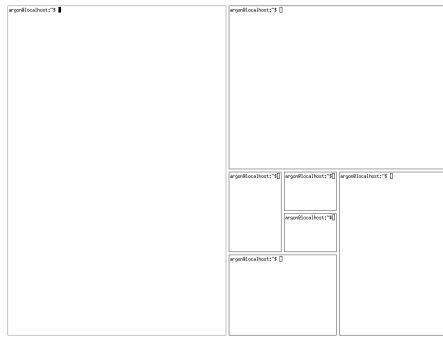
Verschiedene Arten des Tilings lassen sich in dieser Gruppe zusammenfassen, denn es lässt sich eindeutig allein aus der Position der Fenster in einer Liste, unabhängig von der tatsächlichen Implementation, die Platzierung des jeweiligen Fensters ermitteln. Dadurch ist es grundsätzlich möglich zwischen verschiedenen Darstellungsarten zu wechseln.

### Spiralmodus und Variationen

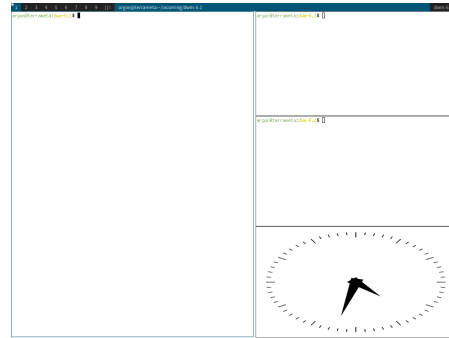
Im *Spiralmodus*, wie in Abbildung 3a zu sehen, oder Variationen wie dem Goldenen Schnitt oder Dwindle werden die Fenster in Form des namensgebenden Muster oder Algorithmus automatisch angeordnet.

### Spalten-/Rastermodus

Die Fenster werden gleichmäßig in einer festen Anzahl von Spalten oder Zeilen in einem Raster (engl. grid) angeordnet.



(a) *Spiraldarstellung von Fenstern mit bspwm[12]*



(b) *dwm[13] im vertikalen Master & Stack-Modus*

Abbildung 3

## Master & Stacked mode

Der Master & Stacked-Modus (Abbildung 3b) ist eine Variante des Spalten-/Rastermodus bei dem der Bildschirm in zwei Bereiche unterteilt wird. Den Hauptarbeitsbereich *Master*, und den Nebearbeitsbereich *Stack*. Hierbei finden sich im Master-Bereich eine feste Anzahl von Fenstern und im anderen Bereich die restlichen Fenster.

### 1.2.3. Baum-basierende Window-Manager

Diese Art von Window-Managern speichert die Informationen der einzelnen verwalteten Fenster in einer Baumstruktur, die unmittelbar Einfluss auf die graphische Darstellung hat. So können einzelne Bereiche wieder entsprechend unterteilt werden.

## 2. Entscheidung

Ich habe mich für ein Baum-basiertes Layout ausgewählt. Dies bietet eine hohe Flexibilität bei gleichzeitig hohem Strukturierungsgrad. Alle listenbasierende Tiling-Varianten können hier auch implementiert werden. Eine höhere Komplexität in der Implementation wird in Kauf genommen. Ein Constraint-basiertes Modell wurde abgelehnt, da dies zu einer zu willkürlichen Fensterverteilung führen würde und die Implementation als zu schwierig und aufwändig erachtet wurde.

### 3. Projektumgebung

Die Entwicklungsumgebung orientierte sich an der bereits Vorhandenen in der lokalen, unter Linux laufenden Ausführung. Sie bestand aus gcc 6.2.1 als Compiler und Linker, make 4.2.1 zum Automatisieren des Compiler- und Linkprozesses und git 2.10.0 zur Versionskontrolle. Zur statischen Analyse des Quellcodes wurden clang 3.8.1 (mit scan-build make) verwendet, zum Abstimmen der einzufügenden Headerdateien include-what-you-use 0.6. Zum Bearbeiten des Quellcodes wurde der Editor vim 7.4 verwendet.

Tests wurden mit Hilfe des X-Servers Xephyr absolviert. Dies ist ein vollständiger X-Server der in ein Fenster eines bereits gestarteten X-Servers zeichnet.

### 4. Projektverlauf

Nach der Entscheidung einen Tree-basierenden Tiling-Modus zu entwickeln, musste zunächst eine Grundstruktur für den Baum designt und implementiert werden da sich der *wmwm* bis dahin mit einer doppelt-verketteten Liste organisierte (`list.c` und `list.h`).

#### 4.1. Vorarbeiten

Diese bisherige Listenimplementation hatte generische Namen für die Typen und Funktionen. Aus Gründen der Übersichtlichkeit wurden diese zunächst umbenannt. Aus dem Knotentyp `item_t` wurde `list_t` und jeder Listenfunktion wurde das Präfix `list_` vorangesetzt. Die Workspaces (0-9) bleiben weiterhin in einem Array fester Größe gespeichert.

#### 4.2. Baumstruktur

Der Bildschirm wird mit Tiling-Bereichen unterteilt, in einem Bereich sollen sich die Fenster den verfügbaren Platz teilen. Jedes Fenster soll wieder geteilt werden können. Die Abbildung 4a zeigt ein Beispiel mit zwei Tiling-Knoten und insgesamt vier Fenstern, die entsprechende Baumdarstellung wird in Abbildung 4b illustriert.

Die kreisrunden Knoten mit Zahlen repräsentieren einzelne Fenster. Dreieckige Knoten sind Tilingknoten, sie enthalten die Fensterknoten als Kindknoten. Fensterknoten können keine Kindknoten enthalten. Zur Implementation mit mehreren Fenstern unter einem Tilingbereich, muss der Tilingknoten beliebig viele Kindknoten haben können. Dies wurde über eine Variation eines k-adischen Baumes implementiert (vgl. Abbildung 5). Anstatt die Kindknoten in den Tiling-Knoten (Elterknoten) zu speichern, wird nur der erste Kindknoten im Tilingknoten gespeichert, die Kindknoten verweisen dann auf ihre nächsten

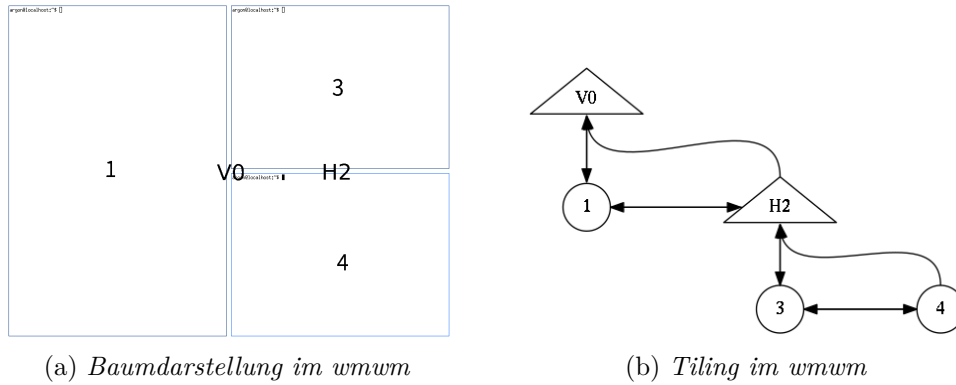


Abbildung 4

Geschwister. So kann der Baum ähnlich einem binären Baum durchlaufen werden, mit dem Kindknoten als linken Knoten und dem Next-Knoten als rechten Knoten. Darüber hinaus wurde in `tree.h` (siehe Unterabschnitt A.1) grundlegende Funktionen zur Arbeit mit diesen Knoten implementiert um die Komplexität der Struktur zu verstecken und somit eine konsistente Struktur zu gewährleisten.

Um die verschiedenen Arten von Knoten im gleichen Baum zu realisieren wurde ein Meta-Struktur namens `container_t` (Abbildung 6) implementiert.

Die Struktur `container_t` enthält zunächst den Typen des Containers. Abhängig vom Typ werden dann unterschiedliche Daten gespeichert. Beim Fenstercontainer `CONTAINER_CLIENT` wird ein Pointer zu einer Client-Struktur (`client_t`) gespeichert, diese enthält alle nötigen Informationen über das Fenster die der Window-Manager zum Verwalten dessen benötigt. Sie selbst wurde nur geringfügig angepasst. Bei einem Tiling-Container `CONTAINER_TILING` werden Tiling-Art und die Anzahl der Unterknoten (`tiles`) gespeichert. Ein Knoten mit `container_t*` als data wird `wtree_t` (von "Window-Tree") genannt. Funktionen zum Arbeiten mit diesen Knoten wurden in

```
typedef struct tree_item tree_t;

struct tree_item {
    void *data; /* arbitrary data */

    tree_t *parent; /* parent node */
    tree_t *prev; /* previous sibling */
    tree_t *next; /* next sibling */
    tree_t *child; /* child */
};
```

Abbildung 5: Knoten der  $k$ -stufigen Baumstruktur (`tree.h`)



```

typedef enum container_types {
    CONTAINER_TILING,
    CONTAINER_CLIENT
} container_type;

typedef struct container {
    container_type type;
    union {
        struct {
            tiling_t tile;
            uint16_t tiles;
        };
        client_t *client;
    };
} container_t;

```

Abbildung 6: *container\_t struct (container\_tree.h)*

window\_tree.h (Unterabschnitt A.2) implementiert.

### 4.3. Integration der Tiling-Funktionalität

Zunächst wurden die vorherigen Verweise auf Positionen in der Fensterliste eines Workspaces mit den Baumknoten `wtree_t` ersetzt. Dann wurden die wichtigen Funktionen `find_client()` und `find_clientp()` an die Baumstruktur angepasst. Diese Funktionen dienen dazu zu einer X-Fensternummer das passende `client_t`-struct zuzuordnen. Immer wenn der X-Server dem Window-Manager etwas über eine Aktion eines Fensters mitteilt, gibt er diese Nummer an. Um dazu die passenden Informationen im struct zu finden muss der Fenster-Baum durchlaufen werden. Zum durchlaufen wurden zwei Funktionen implementiert die den Baum in Pre-Order-Reihenfolge abschreiten.

Die Funktion `wtree_traverse_clients` führt auf jeden Fensterknoten eine generische Funktion aus und die Funktion `wtree_find_client` nutzt eine generische Funktion um einen bestimmten Fensterknoten zu finden (siehe Definition in `window_tree.h`/Unterabschnitt A.2). Die Funktion `wtree_traverse_clients` wurde dort eingesetzt wo Aktionen auf alle Fenster des Workspaces ausgeführt werden mussten, so z.B. beim Verstecken und Anzeigen von Fenstern beim Wechseln des Workspaces.

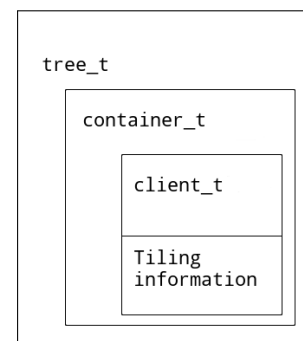


Abbildung 7: *Hierarchie der Strukturen*

Im nächsten Schritt musste die Funktion `set_workspace()` (in `wmwm.c`), die die Zugehörigkeit eines Fensters zu einem Workspace festlegt, so angepasst werden, dass die Fenster an den richtigen Knoten angehängen werden.

Eine der fundamentalen neuen Funktionen im Hauptprogramm ist `update_clues()`, hier werden die Größen und Position der einzelnen Bereiche und Fenster festgelegt. Durch die Baumstruktur fällt dies relativ einfach aus. Die Funktion arbeitet rekursiv, sie terminiert immer wenn ihr ein Fensterknoten übergeben wird, dann wird die Größe festgelegt. Sie beginnt mit der Größe des Bildschirms (in Pixeln) und ruft sie für jeden Unterbereich mit der zustehenden Größe (Restgröße durch Anzahl der Knoten (ohne Floatingknoten)) auf. Dabei wird auch die Position angepasst.

Desweiteren musste die Funktion `update_geometry()` angepasst werden, hier werden die Größen- und Positionsänderungen der einzelnen Fenster tatsächlich umgesetzt. Da bei Tilingfenstern der Window-Manager die Positionierung übernimmt, kann in dem Fall die Anpassung an die Wünsche des jeweiligen Programmes übersprungen werden.

Zuletzt wurden noch zusätzliche Tastenkombinationen zum Ändern des Tilingmodus für das nächste öffnende Fenster und für Fenster die sich bereits in einem Container befinden hinzugefügt. Für neue Fenster gibt es die globale Variable `tiling_mode`, bei vorhandenen Fenstern werden `toggle_floating()` und `toggle_tiling()` aufgerufen. Sie setze wiederum die Eigenschaften in den einzelnen Knoten und lösen ein Neuzeichnen des Teilbaumes aus.

## 5. Literaturverzeichnis

- [1] Robert W. Scheifler. *RFC 1013: X WINDOW SYSTEM PROTOCOL, VERSION 11*. Techn. Ber. Apr. 1987, S. 1–101. URL: <https://tools.ietf.org/html/rfc1013> (besucht am 22.09.2016).
- [2] Brad A. Myers. „A taxonomy of window manager user interfaces“. In: *IEEE Computer Graphics and Applications* 8.5 (Sep. 1988), S. 65–84.
- [3] Michael Cardell Widerkrantz. *mcwm - a minimalist window manager*. 2016. URL: <http://hack.org/mc/projects/mcwm/> (besucht am 22.09.2016).
- [4] Ciaran Anscorb. *evilwm - a minimalist window manager for the X Windows System*. 2015. URL: <http://www.6809.org.uk/evilwm> (besucht am 22.09.2016).
- [5] Nick Welch. *tinywm*. 2005. URL: <http://incise.org/tinywm.html> (besucht am 22.09.2016).
- [6] Michael Cardell Widerkrantz. *History of mcwm*. URL: <http://hack.org/mc/projects/mcwm/history.html> (besucht am 22.09.2016).
- [7] Ellis S. Cohen, Edward T. Smith und Lee A. Iverson. „Constrain-based tile windows“. In: *IEEE Computer Graphics and Applications* 6.5 (Mai 1986), S. 35–45.
- [8] Saul Greenberg, Murray Peterson und Ian H. Witten. „Issues and experiences in the design of a window management system“. In: *Canadian Processing Society Edmonton Conference*. Sep. 1986, S. 33–44.
- [9] X Desktop Group - freedesktop.org. *Extended Window Manager Hints*. 29. Nov. 2011. URL: <https://standards.freedesktop.org/wm-spec/wm-spec-latest.html> (besucht am 22.09.2016).
- [10] David Rosenthal und Stuart W. Marks. *Inter-Client Communication Conventions Manual*. 1994. URL: <https://tronche.com/gui/x/icccm/sec-4.html> (besucht am 22.09.2016).
- [11] Blane A. Bell und Steven K. Feiner. „Dynamic Space Management of User Interfaces“. In: *ACM Symp. on User Interface Software and Technology*. 2000.
- [12] Bastien Dejean. *bspwm github repository*. URL: <https://github.com/baskerville/bspwm> (besucht am 22.09.2016).
- [13] Anselm R. Garbe. *suckless.org dwm - dynamic window manager*. 2015. URL: <http://dwm.suckless.org> (besucht am 22.09.2016).

## 6. Glossar

**Fork** eine Abspaltung von einem Softwareprojekt.

**git** Versionsverwaltung zur Erfassung von Änderungen an Dateien.

**Linux** ein freier Betriebssystemskernel, oft synonym für komplette Linuxdistribution verwendet.

**repository** verwaltetes (versioniertes) Verzeichnis mit Quelltext.

**Tiling** (engl. Fliesenlegen) das Ausfüllen des Bildschirms mit Fenstern die nicht überlappen.

**Window-Manager** ein Programm zur Organisation und Behandlung von Fenstern auf einer graphischen Oberfläche.

**X11** X in der Version 11.

## A. Anlagen

### A.1. tree.h

```
#ifndef __WMWM__TREE_H__
#define __WMWM__TREE_H__

/* n-ary Tree implementation
 *
 * Each node has one parent, two siblings (prev, next), one child
 * and a pointer to arbitrary data.
 *
 * child is always the first (prev == NULL) in the list of siblings
 */
typedef struct tree_item tree_t;
struct tree_item {
    void *data; /* arbitrary data */

    tree_t *parent; /* parent node */
    tree_t *prev; /* previous sibling */
    tree_t *next; /* next sibling */
    tree_t *child; /* child */ /* only for workspace and tiling! */
};

/* helper functions */
static tree_t *tree_parent(tree_t *node) { return node->parent; }
static tree_t *tree_child(tree_t *node) { return node->child; }
static void *tree_data(tree_t *node) { return node->data; }

/* create new node */
tree_t *tree_new(tree_t *parent, tree_t *prev, tree_t *next, tree_t *child,
    void *data);

/* remove node from tree, fix ancestors, take child with it */
void tree_extract(tree_t *node);
/* insert _node_ as sibling before _old_ */
void tree_insert(tree_t *next, tree_t *node);
/* insert _node_ as sibling after _old_ */
void tree_add(tree_t *next, tree_t *node);
/* replace _from_ with _to_ */
void tree_replace(tree_t *from, tree_t *to);

#endif /* __WMWM__TREE_H__ */
```

## A.2. window\_tree.h

```
#ifndef __WMWM__CONTAINER_TREE_H__
#define __WMWM__CONTAINER_TREE_H__
#include <stdint.h>    // for uint16_t
#include "stdbool.h"   // for bool
#include "tree.h"      // for tree_t
#include "wmwm.h"      // for client_t

typedef tree_t wtree_t;

/* Tiling modes */
typedef enum tiling_modes {
    TILING_HORIZONTAL,
    TILING_VERTICAL,
    TILING_FLOATING
} tiling_t;

/* general container types */
typedef enum container_types {
    CONTAINER_TILING,
    CONTAINER_CLIENT
} container_type;

/* container * static local helper functions in window_tree.c */
typedef struct container {
    container_type type;
    union {
        struct {
            tiling_t tile;
            uint16_t tiles;
        };
        client_t *client;
    };
} container_t;

/* create new node with client/tiling "container" */
wtree_t* wtree_new_client(client_t *client);
wtree_t* wtree_new_tiling(tiling_t tile);

/* free node and its data */
void wtree_free(wtree_t *node);

/* get client from node */
```

```

client_t *wtree_client(wtree_t *node);

/* check if node is ... */
bool wtree_is_client(wtree_t *node);
bool wtree_is_tiling(wtree_t *node);
bool wtree_is_singleton(wtree_t* child);

/* get number of tiles/children */
uint16_t wtree_get_tiles(wtree_t *node);

/* get tiling_t of node/parent */
tiling_t wtree_tiling(wtree_t *node);
tiling_t wtree_parent_tiling(wtree_t *node);

/* set tiling_t of node/parent */
void wtree_set_tiling(wtree_t *node, tiling_t tiling);
void wtree_set_parent_tiling(wtree_t *node, tiling_t tiling);

/* add/append sibling/children */
void wtree_add_sibling(wtree_t *current, wtree_t *node);
void wtree_append_sibling(wtree_t *current, wtree_t *node);
void wtree_append_child(wtree_t *parent, wtree_t *node);

/* put new tiling node between client and client->parent */
void wtree_inter_tile(wtree_t *client, tiling_t mode);

/* add tiling-node as sibling to current and client-node as child to tiling-
   node */
void wtree_add_tile_sibling(wtree_t *current, wtree_t *node, tiling_t tiling)
    ;

/* unlink node from tree, fix siblings and parent */
void wtree_remove(wtree_t *node);

/* for each client-node below node, do action(client), pre-order*/
void wtree_traverse_clients(wtree_t *node, void(*action)(client_t *));
/* find node below _node_ that has compare(client) == true, pre-order */
client_t *wtree_find_client(wtree_t *node, bool(*compare)(client_t*, void *),
    void *arg);

/* print tree to dot file */
void wtree_print_tree(wtree_t *cur);

#endif

```

## B. Kundendokumentation

### B.1. Einleitung

Der Window-Manager *wmwm* wurde um eine Tiling-Funktionalität erweitert. Hierbei kann nun jedes Fenster einen der folgenden zwei Zustände haben:

- **Floating-Modus**

Diese Fenster verhalten sich wie zuvor, das Fenster kann frei bewegt und in der Größe verändert werden. Diese Fenster sind immer im Vordergrund vor den Fenstern im Tiling-Modus.

- **Tiling-Modus**

Das Fenster wird in der automatischen Anordnung integriert. Fenster sind in Gruppen organisiert.

Im **Tiling-Modus** kann jede Gruppe beliebig viele Fenster und Untergruppen enthalten, innerhalb dieser Gruppe werden die Fenster gleichmäßig in ihrer Größe aufgeteilt. Diese Aufteilung kann in **vertikal** oder **horizontal** geschehen.

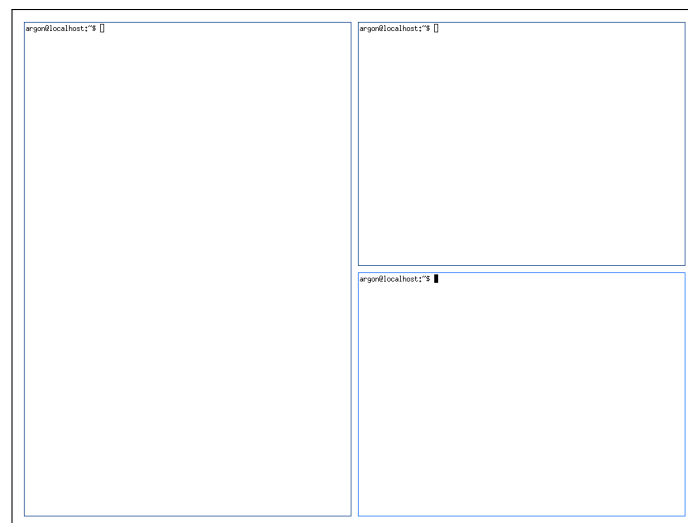


Abbildung 1: *wmwm* mit drei Fenstern im Tiling-Modus

In Abbildung 1 ist ein beispielhaft ein Arbeitsbereich mit drei Fenstern dargestellt. Die beiden Fenster auf der rechten Seite sind in einer Gruppe mit **horizontalem Tiling** zusammengefasst. Diese und das Fenster auf der linken Seite bilden wiederum eine Gruppe mit **vertikalem Tiling**.













## B.2. Bedienung

Alle neuen Fenster werden in dieser Gruppe rechts angehängen. Neue Fenster werden immer hinter oder unter dem aktuell **fokussierten** Fenster in der Gruppe angehängen. Es kann die Anordnung live zwischen **vertikalem** und **horizontalem** Tiling gewechselt werden. Wird von **Floating-Modus** zu **Tiling-Modus** gewechselt, dann wird das Fenster in die vorherige Ordnung übernommen. Im umgekehrten Falle, wird die Fläche, den das Fenster in der automatischen Anordnung hatte, an die anderen Fenster freigegeben und das Fenster ausgelöst.

### Tastenkombination

Folgende neue Tastenkombinationen wurden implementiert.

 + 	zwischen vertikalem und horizontalem Tiling-Modus wechseln
 + 	zwischen Floating- und Standard-Tiling-Modus wechseln
  + 	Tiling-Modus der aktuellen Gruppe wechseln
  + 	Aktuelles Fenster zwischen Floating- und Tiling-Modus wechseln

## B.3. Konfiguration

### Konfigurationsdatei

Die Konfiguration wird verbleibt in Quelltext der Datei `config.h`, dort gibt es zusätzliche als Macro definierte Optionen:

DEFAULT_TILING_MODE	TILING_VERTICAL (Standard)    Vertikales Tiling TILING_HORIZONTAL                Horizontales Tiling TILING_FLOATING                  Floating-Fenster
GAPWIDTH	Abstand zwischen Fenstern in Pixel (5)
USERKEY_TILING	Taste für Tiling-Einstellung (XK_V)
USERKEY_FLOATING	Taste für Floating-Einstellung (XK_F)

### Kommandozeilenargumente

Eine zusätzliche Kommandozeilenoptionen wurden implementiert: Mit

`-g PIXEL`

kann der Abstand zwischen den Tilingfenstern beim Start des Programms angegeben werden.