```
//-- Key binds ---------------------------------------------------------------
---------
#define MODKEY Mod4Mask // Win key
#define TAGKEYS(KEY,TAG) \
{MODKEY,              KEY,        use_tag,       {.i = TAG} }, \
{MODKEY|ShiftMask,    KEY,        move_tag,      {.i = TAG} },

static KeyBinding keys[] = {
   // modifier                     key         function       argument
   { MODKEY|ShiftMask,             XK_Return,  spawn,         {.v = "urxvt"} },
   { MODKEY|ShiftMask,             XK_w,       spawn,         {.v = "chromium"}
},
   { MODKEY|ShiftMask,             XK_g,       spawn,         {.v = "gvim"} },
   { MODKEY|ShiftMask,             XK_t,       spawn,         {.v = "thunar"} },
   { MODKEY|ShiftMask,             XK_r,       spawn,         {.v = "gmrun"} },
   { MODKEY,                       XK_q,       quit,          {0} },
   { MODKEY,                       XK_c,       client_close,  {0} },
   { MODKEY,                       XK_m,       set_layout,    {.v = "max"} },
   { MODKEY,                       XK_v,       set_layout,    {.v = "vertical"} },
   { MODKEY,                       XK_h,       set_layout,    {.v = "horizontal"}
},
   { MODKEY,                       XK_f,       set_pseudotile, {0} },
   { MODKEY|ShiftMask,             XK_v,       split_v,       {.f = 0.5} },
   { MODKEY|ShiftMask,             XK_h,       split_h,       {.f = 0.5} },
   { MODKEY,                       XK_r,       frame_remove,  {0} },
   { MODKEY,                       XK_Left,    focus,         {.v = "left"} },
   { MODKEY,                       XK_Right,   focus,         {.v = "right"} },
   { MODKEY,                       XK_Up,      focus,         {.v = "up"} },
   { MODKEY,                       XK_Down,    focus,         {.v = "down"} },
   { MODKEY,                       XK_comma,   focus_monitor, {.i = 0} },
   { MODKEY,                       XK_period,  focus_monitor, {.i = 1} },
   { MODKEY|ShiftMask,             XK_Left,    shift,         {.v = "left"} },
   { MODKEY|ShiftMask,             XK_Right,   shift,         {.v = "right"} },
   { MODKEY|ShiftMask,             XK_Up,      shift,         {.v = "up"} },
   { MODKEY|ShiftMask,             XK_Down,    shift,         {.v = "down"} },
   { MODKEY,                       XK_Tab,     cycle,         {0} },
   TAGKEYS(                        XK_1,                      0)
   TAGKEYS(                        XK_2,                      1)
   TAGKEYS(                        XK_3,                      2)
   TAGKEYS(                        XK_4,                      3)
   TAGKEYS(                        XK_5,                      4)
   TAGKEYS(                        XK_6,                      5)
   TAGKEYS(                        XK_7,                      6)
   { MODKEY|ControlMask,           XK_Left,    resize_frame,  {.v="left"} },
   { MODKEY|ControlMask,           XK_Right,   resize_frame,  {.v="right"} },
   { MODKEY|ControlMask,           XK_Up,      resize_frame,  {.v="up"} },
   { MODKEY|ControlMask,           XK_Down,    resize_frame,  {.v="down"} },
};

// -- Mouse bindings ---------------------------------------------------------
---------
static MouseBinding buttons[] = {
   //event mask        button         function
   { MODKEY,           Button1,       mouse_function_move },
   { MODKEY,           Button3,       mouse_function_resize },
};
```



```
//-- Appearance and Tags -----------------------------------------------------
---------
static const char font[]               = "-*-clean-medium-r-*-*-12-*-*-*-*-*-*-*";
static const int window_gap            = 3;
static const int snap_distance         = 10;
static const int frame_border_width    = 1;
static const int window_border_width   = 1;
static const int bh                    = 20;  // bar height
static const int systray_width         = 100;

#define NUMCOLORS 4
static const char colors[NUMCOLORS][ColLast][8] = {
   // frame-border   window-border  foreground  background
   { "#000000",      "#000000",     "#000000",  "#F6F6F6"}, // 0 - normal
   { "#000000",      "#1793D0",     "#1793D0",  "#F6F6F6"}, // 1 - selected
   { "#000000",      "#F6F6F6",     "#333333",  "#F6F6F6"}, // 2 - inactive tags
   { "#000000",      "#FF0000",     "#FF0000",  "#F6F6F6"}, // 3 - urgent tags
};

#define NUMTAGS 7
static const char tags[NUMTAGS][10] = { "Eins", "Zwei", "Drei",
                                        "Vier", "FÃ¼nf", "Sechs", "Sieben" };

//-- Other configurations ----------------------------------------------------
---------
static const int focus_follows_mouse = 1;
static const int focus_follows_shift = 1;
static const int focus_new_clients = 1;
static const int raise_on_click = 1;
static const int default_frame_layout  = 0;
static const float resize_step = 0.025;

//-- Rules -------------------------------------------------------------------
---------
static const Rule custom_rules[] = {
   //Condition, Value || Tag, Manage, Pseudotile
   { "windowtype", "_NET_WM_WINDOW_TYPE_DIALOG",        -1, 1, 1 },
   { "windowtype", "_NET_WM_WINDOW_TYPE_UTILITY",       -1, 1, 1 },
   { "windowtype", "_NET_WM_WINDOW_TYPE_SPLASH",        -1, 1, 1 },
   { "windowtype", "_NET_WM_WINDOW_TYPE_NOTIFICATION", -1, 0, 0 },
   { "windowtype", "_NET_WM_WINDOW_TYPE_DOCK",          -1, 0, 0 },
   { "class",  "Gmrun",                                -1, 1, 1 },
   { "class",  "Canvas",                               -1, 1, 1 },
   { "class", "net-sourceforge-jnlp-runtime-Boot",      -1, 1, 1 },
};
```

```
/*
 * FusionWM - clientlist.h
 */

#ifndef _CLIENTLIST_H_
#define _CLIENTLIST_H_

#include <X11/Xproto.h>
#include <X11/Xutil.h>
#include <X11/Xatom.h>
#include <glib.h>
#include <stdbool.h>

#include "layout.h"

#define ENUM_WITH_ALIAS(Identifier, Alias) Identifier, Alias = Identifier

#define _NET_WM_STATE_REMOVE        0    /* remove/unset property */
#define _NET_WM_STATE_ADD           1    /* add/set property */
#define _NET_WM_STATE_TOGGLE        2    /* toggle property  */

enum {
    NetSupported = 0,
    NetActiveWindow,
    NetWmName,
    NetWmWindowType,
    NetWmState,
    /* window states */
    NetWmStateFullscreen,
    /* window types */
    ENUM_WITH_ALIAS(NetWmWindowTypeDesktop, NetWmWindowTypeFIRST),
    NetWmWindowTypeDock,
    NetWmWindowTypeUtility,
    NetWmWindowTypeSplash,
    NetWmWindowTypeDialog,
    NetWmWindowTypeNotification,
    ENUM_WITH_ALIAS(NetWmWindowTypeNormal, NetWmWindowTypeLAST),
    /* the count of hints */
    NetCOUNT
};

struct HSTag;
struct HSClient;

Atom g_netatom[NetCOUNT];
extern char* g_netatom_names[];

typedef struct HSClient {
    Window      window;
    XRectangle  last_size;
    HSTag*      tag;
    XRectangle  float_size;
    char        title[256];  // This is never NULL
    bool        urgent;
    bool        fullscreen;
    bool        pseudotile; // should be "floating"
} HSClient;

typedef struct {
   char *condition;
   char *cond_value;
   int   tag;
   int   manage;
   int   pseudotile;
} Rule;

//---
void clientlist_init();
void clientlist_destroy();

void window_focus(Window window);
void window_unfocus(Window window);
void window_unfocus_last();
```



```
// adds a new client to list of managed client windows
HSClient* manage_client(Window win);
void unmanage_client(Window win);

// destroys a special client
void destroy_client(HSClient* client);

HSClient* get_client_from_window(Window window);
HSClient* get_current_client();
XRectangle client_outer_floating_rect(HSClient* client);

void client_setup_border(HSClient* client, bool focused);
void client_resize(HSClient* client, XRectangle rect);
void client_resize_floating(HSClient* client, HSMonitor* m);
void client_resize_fullscreen(HSClient* client, HSMonitor* m);
void client_clear_urgent(HSClient* client);
void client_update_wm_hints(HSClient* client);
void client_update_title(HSClient* client);
void client_close(const Arg *arg);

void client_set_fullscreen(HSClient* client, bool state);
void client_set_pseudotile(HSClient* client, bool state);
void set_pseudotile(const Arg* arg);

void window_set_visible(Window win, bool visible);

// set the desktop property of a window
void ewmh_handle_client_message(XEvent* event);

void rules_apply(struct HSClient* client, int *manage);

#endif
```

```c
/*
 * FusionWM - globals.h
 */

#ifndef _GLOBALS_H_
#define _GLOBALS_H_

#include <glib.h>
#include <stddef.h>
#include <stdbool.h>
#include <locale.h>
#include <X11/Xlib.h>
#include <X11/Xatom.h>

#define LENGTH(X) (sizeof(X)/sizeof(*X))
#define MOD(X, N) ((((X) % (signed)(N)) + (signed)(N)) % (signed)(N))
#define ATOM(A) XInternAtom(g_display, (A), False)

#define WINDOW_MIN_HEIGHT 32
#define WINDOW_MIN_WIDTH 32
#define FRAME_MIN_FRACTION 0.1
#define STRING_BUF_SIZE 256

#define RECTANGLE_EQUALS(a, b) (\
        (a).x == (b).x && (a).y == (b).y && \
        (a).width == (b).width && (a).height == (b).height )

#define ROOT_EVENT_MASK   (PropertyChangeMask|SubstructureRedirectMask|SubstructureNotifyMask|ButtonPressMask|EnterWindowMask|LeaveWindowMask|StructureNotifyMask)
#define CLIENT_EVENT_MASK (EnterWindowMask|FocusChangeMask|PropertyChangeMask|StructureNotifyMask)
#define CLEANMASK(mask)   (mask & ~(numlockmask|LockMask) & (ShiftMask|ControlMask|Mod1Mask|Mod2Mask|Mod3Mask|Mod4Mask|Mod5Mask))

Display*    g_display;
int         g_screen;
Window      g_root;
int         g_screen_width;
int         g_screen_height;
bool        g_aboutToQuit;

typedef union {
   int i;
   unsigned int ui;
   float f;
   const void *v;
} Arg;

//---
void die(const char *errstr, ...);
unsigned long getcolor(const char *colstr);
bool gettextprop(Window w, Atom atom, char *text, unsigned int size);
void spawn(const Arg *arg);
void quit(const Arg *arg);

#endif
```



```c
/*
 * FusionWM - inputs.h
 */

#ifndef _INPUTS_H_
#define _INPUTS_H_

#include <glib.h>
#include <X11/Xlib.h>

enum SnapFlags {
    // which edges are considered to snap
    SNAP_EDGE_TOP       = 0x01,
    SNAP_EDGE_BOTTOM    = 0x02,
    SNAP_EDGE_LEFT      = 0x04,
    SNAP_EDGE_RIGHT     = 0x08,
    SNAP_EDGE_ALL       =
        SNAP_EDGE_TOP | SNAP_EDGE_BOTTOM | SNAP_EDGE_LEFT | SNAP_EDGE_RIGHT,
};

// foreward declarations
struct HSClient;
struct HSTag;

typedef void (*MouseFunction)(XMotionEvent*);

typedef struct MouseBinding {
    unsigned int mask;
    unsigned int button;
    MouseFunction function;
} MouseBinding;

typedef struct KeyBinding {
   unsigned int mod;
   KeySym keysym;
   void (*func)(const Arg *);
   const Arg arg;
} KeyBinding;

//---
void inputs_init();
void inputs_destroy();

void key_find_binds(char* needle, GString** output);
MouseBinding* mouse_binding_find(unsigned int modifiers, unsigned int button);
void grab_keys();
void grab_buttons();

void mouse_start_drag(XEvent* ev);
void mouse_stop_drag();
void handle_motion_event(XEvent* ev);

// get the vector to snap a client to it's neighbour
void client_snap_vector(struct HSClient* client, struct HSTag* tag,
                        enum SnapFlags flags,
                        int* return_dx, int* return_dy);

/* some mouse functions */
void mouse_function_move(XMotionEvent* me);
void mouse_function_resize(XMotionEvent* me);

void key_press(XEvent* ev);
void update_numlockmask();

#endif
```

```
/*
 * FusionWM - layout.h
 */

#ifndef _LAYOUT_H_
#define _LAYOUT_H_

#include <X11/cursorfont.h>
#include "globals.h"

#include <glib.h>
#include <stdlib.h>

enum {
    ALIGN_VERTICAL = 0,
    ALIGN_HORIZONTAL,
};

enum {
    LAYOUT_VERTICAL = 0,
    LAYOUT_HORIZONTAL,
    LAYOUT_MAX,
    LAYOUT_COUNT,
};

enum {
    TYPE_CLIENTS = 0,
    TYPE_FRAMES,
};

enum { ColFrameBorder, ColWindowBorder, ColFG, ColBG, ColLast }; /* color */

struct HSClient;
struct HSFrame;
struct HSTag;

typedef int (*ClientAction)(struct HSClient*, void* data);

#define FRACTION_UNIT 10000

typedef struct HSLayout {
    int align;         // ALIGN_VERTICAL or ALIGN_HORIZONTAL
    struct HSFrame* a; // first child
    struct HSFrame* b; // second child
    int selection;
    int fraction; // size of first child relative to whole size
} HSLayout;

typedef struct HSFrame {
    union {
        HSLayout layout;
        struct {
            Window* buf;
            size_t  count;
            size_t  floatcount;
            int     selection;
            int     layout;
        } clients;
    } content;
    int type;
    struct HSFrame* parent;
    Window window;
    bool   window_visible;
} HSFrame;

typedef struct HSMonitor {
    struct HSTag*      tag;    // currently viewed tag
    struct {
        // last saved mouse position
        int x;
        int y;
    } mouse;
    XRectangle  rect;   // area for this monitor
```



```
    Window barwin;
    int primary;
} HSMonitor;

typedef struct HSTag {
    GString*    name;   // name of this tag
    HSFrame*    frame;  // the master frame
    int         flags;
    bool       urgent;
} HSTag;

// globals
GArray*     g_tags; // Array of HSTag*
GArray*     g_monitors; // Array of HSMonitor
int         g_cur_monitor;
HSFrame*    g_cur_frame; // currently selected frame
bool        g_tag_flags_dirty;
extern char* g_layout_names[];

//--- Functions
void layout_init();
void layout_destroy();

// for frames
HSFrame* frame_create_empty();
void frame_insert_window(HSFrame* frame, Window window);
HSFrame* frame_current_selection();
bool frame_remove_window(HSFrame* frame, Window window);
void frame_destroy(HSFrame* frame, Window** buf, size_t* count);
void frame_split(HSFrame* frame, int align, int fraction);
void split_v(const Arg *arg);
void split_h(const Arg *arg);
void resize_frame(const Arg *arg);

void frame_apply_layout(HSFrame* frame, XRectangle rect);

void cycle(const Arg * arg);

HSFrame* frame_neighbour(HSFrame* frame, char direction);
int frame_inner_neighbour_index(HSFrame* frame, char direction);
void focus(const Arg* arg);

int frame_focus_recursive(HSFrame* frame);
void frame_do_recursive(HSFrame* frame, void (*action)(HSFrame*), int order);
void frame_show_clients(HSFrame* frame);
int frame_foreach_client(HSFrame* frame, ClientAction action, void* data);

void set_layout(const Arg *arg);

Window frame_focused_window(HSFrame* frame);
bool frame_focus_window(HSFrame* frame, Window win);
bool focus_window(Window win, bool switch_tag, bool switch_monitor);
void shift(const Arg *arg);
void frame_remove(const Arg *arg);
void frame_set_visible(HSFrame* frame, bool visible);

// for tags
void add_tag(const char* name);
HSTag* find_tag(const char* name);
void move_tag(const Arg *arg);
void tag_move_window(HSTag* target);

// for monitors
HSMonitor* monitor_with_frame(HSFrame* frame);
HSMonitor* find_monitor_with_tag(HSTag* tag);
void add_monitor(XRectangle rect, HSTag* tag, int primary);
void monitor_focus_by_index(int new_selection);
int monitor_index_of(HSMonitor* monitor);
void focus_monitor(const Arg *arg);
HSMonitor* get_current_monitor();
void monitor_set_tag(HSMonitor* monitor, HSTag* tag);
void use_tag(const Arg *arg);
void monitor_apply_layout(HSMonitor* monitor);
```

```
    void all_monitors_apply_layout();
    void monitors_init();

150 // for bars and miscellaneous
    void create_bar(HSMonitor *mon);
    void draw_bar(HSMonitor *mon);
    void draw_bars();

155 void updatestatus(void);
    void drawtext(const char *text, unsigned long col[ColLast]);
    void initfont(const char *fontstr);
    int textnw(const char *text, unsigned int len);
    int get_textw(const char *text);
160
    HSMonitor* wintomon(Window w);

    #endif
```



```
    /*
     * FusionWM - clientlist.c
     */

5   #include "clientlist.h"
    #include "globals.h"
    #include "layout.h"
    #include "inputs.h"
    #include "config.h"
10
    #include <glib.h>
    #include <assert.h>
    #include <stdio.h>
    #include <sys/types.h>
15  #include <string.h>

    GHashTable* g_clients; // container of all clients
    unsigned long wincolors[NUMCOLORS][ColLast];

20  enum { WMProtocols, WMDelete, WMState, WMTakeFocus, WMLast }; /* default atoms *
    /
    static Atom g_wmatom[WMLast];

    static HSClient* create_client() {
        HSClient* hc = g_new0(HSClient, 1);
25      hc->urgent = false;
        hc->fullscreen = false;
        hc->pseudotile = false;
        return hc;
    }
30
    /* list of names of all _NET-atoms */
    char* g_netatom_names[NetCOUNT] = {
        [ NetSupported                ] = "_NET_SUPPORTED"                    ,
        [ NetActiveWindow             ] = "_NET_ACTIVE_WINDOW"                ,
35      [ NetWmName                   ] = "_NET_WM_NAME"                      ,
        [ NetWmWindowType             ] = "_NET_WM_WINDOW_TYPE"               ,
        [ NetWmState                  ] = "_NET_WM_STATE"                     ,
        [ NetWmStateFullscreen        ] = "_NET_WM_STATE_FULLSCREEN"          ,
        [ NetWmWindowTypeDesktop      ] = "_NET_WM_WINDOW_TYPE_DESKTOP"       ,
40      [ NetWmWindowTypeDock         ] = "_NET_WM_WINDOW_TYPE_DOCK"          ,
        [ NetWmWindowTypeUtility      ] = "_NET_WM_WINDOW_TYPE_UTILITY"       ,
        [ NetWmWindowTypeSplash       ] = "_NET_WM_WINDOW_TYPE_SPLASH"        ,
        [ NetWmWindowTypeDialog       ] = "_NET_WM_WINDOW_TYPE_DIALOG"        ,
        [ NetWmWindowTypeNotification ] = "_NET_WM_WINDOW_TYPE_NOTIFICATION",
45      [ NetWmWindowTypeNormal       ] = "_NET_WM_WINDOW_TYPE_NORMAL"        ,
    };

    /// TYPES ///
    typedef struct {
50      char*   name;
        bool    (*matches)(char* cond_value, HSClient* client);
    } HSConditionType;

    /// DECLARATIONS ///
55  static bool condition_class(char* rule, HSClient* client);
    static bool condition_title(char* rule, HSClient* client);
    static bool condition_windowtype(char* rule, HSClient* client);

    /// GLOBALS ///
60  static HSConditionType g_condition_types[] = {
        {   "class",     condition_class },
        {   "title",     condition_title },
        {   "windowtype", condition_windowtype },
    };
65
    //------
    void ewmh_update_active_window(Window win) {
        XChangeProperty(g_display, g_root, g_netatom[NetActiveWindow],
            XA_WINDOW, 32, PropModeReplace, (unsigned char*)&(win), 1);
70  }

    void ewmh_handle_client_message(XEvent* event) {
```

```
    XClientMessageEvent* me = &(event->xclient);
    int index;
    for (index = 0; index < NetCOUNT; index++) {
        if (me->message_type == g_netatom[index])
            break;
    }
    if (index >= NetCOUNT) return;

    HSClient* client;

    switch (index) {
        case NetActiveWindow:
            // only steal focus it allowed to the current source
            // (i.e.  me->data.l[0] in this case as specified by EWMH)
            if (me->data.l[0] == 2) {
                focus_window(me->window, true, true);
            }
            break;
        case NetWmState:
            client = get_client_from_window(me->window);
            if (!client) break;

            /* mapping between EWMH atoms and client struct members */
            struct {
                int     atom_index;
                bool    enabled;
                void    (*callback)(HSClient*, bool);
            } client_atoms[] = {
                { NetWmStateFullscreen,
                    client->fullscreen,     client_set_fullscreen },
            };

            /* me->data.l[1] and [2] describe the properties to alter */
            for (int prop = 1; prop <= 2; prop++) {
                if (me->data.l[prop] == 0) continue;

                /* check if we support the property data[prop] */
                int i;
                for (i = 0; i < LENGTH(client_atoms); i++) {
                    if (g_netatom[client_atoms[i].atom_index] == me->data.l[prop
])
                        break;
                }
                if (i >= LENGTH(client_atoms))  continue;

                bool new_value[] = {
                    [ _NET_WM_STATE_REMOVE  ] = false,
                    [ _NET_WM_STATE_ADD     ] = true,
                    [ _NET_WM_STATE_TOGGLE  ] = !client_atoms[i].enabled,
                };
                int action = me->data.l[0];

                /* change the value */
                client_atoms[i].callback(client, new_value[action]);
            }
            break;
        default:
            break;
    }
}
//----
void clientlist_init() {
    g_wmatom[WMProtocols] = XInternAtom(g_display, "WM_PROTOCOLS", False);
    g_wmatom[WMDelete] = XInternAtom(g_display, "WM_DELETE_WINDOW", False);
    g_wmatom[WMState] = XInternAtom(g_display, "WM_STATE", False);
    g_wmatom[WMTakeFocus] = XInternAtom(g_display, "WM_TAKE_FOCUS", False);
    // init actual client list
    g_clients = g_hash_table_new_full(g_int_hash, g_int_equal,
                                      NULL, (GDestroyNotify)destroy_client);
    //init colors
    for(int i=0; i<NUMCOLORS; i++){
       wincolors[i][ColFrameBorder] = getcolor(colors[i][ColFrameBorder]);
       wincolors[i][ColWindowBorder] = getcolor(colors[i][ColWindowBorder]);
```



```
       wincolors[i][ColFG] = getcolor(colors[i][ColFG]);
       wincolors[i][ColBG] = getcolor(colors[i][ColBG]);
    }

    /* init ewmh net atoms */
    for (int i = 0; i < NetCOUNT; i++) {
        if (g_netatom_names[i] == NULL) {
            g_warning("no name specified in g_netatom_names for atom number %d\n", i);
            continue;
        }
        g_netatom[i] = ATOM(g_netatom_names[i]);
    }

    /* tell which ewmh atoms are supported */
    XChangeProperty(g_display, g_root, g_netatom[NetSupported], XA_ATOM, 32,
        PropModeReplace, (unsigned char *) g_netatom, NetCOUNT);
}

void clientlist_destroy() {
    g_hash_table_destroy(g_clients);
}

HSClient* get_client_from_window(Window window) {
    return (HSClient*) g_hash_table_lookup(g_clients, &window);
}

static void window_grab_button(Window win){
   XGrabButton(g_display, AnyButton, 0, win, true, ButtonPressMask,
      GrabModeSync, GrabModeSync, None, None);
}

HSClient* manage_client(Window win) {
    if (get_client_from_window(win))            return NULL;

    // init client
    HSClient* client = create_client();
    HSMonitor* m = get_current_monitor();
    // set to window properties
    client->window = win;
    client_update_title(client);

    unsigned int border, depth;
    Window root_win;
    int x, y;
    unsigned int w, h;
    XGetGeometry(g_display, win, &root_win, &x, &y, &w, &h, &border, &depth);
    // treat wanted coordinates as floating coords
    //client->float_size.x = x;
    //client->float_size.y = y;
   XRectangle size = client->float_size;
    size.width = w;
    size.height = h;
    size.x = m->rect.x + m->rect.width/2 - size.width/2;
    size.y = m->rect.y + m->rect.height/2 - size.height/2 + bh;
    client->float_size = size;
    client->last_size = size;
   XMoveResizeWindow(g_display, client->window, size.x, size.y, size.width, size
.height);

    // apply rules
    int manage = 1;
    rules_apply(client, &manage);

    if (!manage) {
        destroy_client(client);
        // map it... just to be sure
        XMapWindow(g_display, win);
        return NULL;
    }

    // actually manage it
    g_hash_table_insert(g_clients, &(client->window), client);
    XSetWindowBorderWidth(g_display, win, window_border_width);
```

```
        // insert to layout
        if (!client->tag)   client->tag = m->tag;

        // get events from window
        XSelectInput(g_display, win, CLIENT_EVENT_MASK);
        window_grab_button(win);
        frame_insert_window(client->tag->frame, win);

        monitor_apply_layout(find_monitor_with_tag(client->tag));

        return client;
    }

    void unmanage_client(Window win) {
        HSClient* client = get_client_from_window(win);
        if (!client)   return;

        // remove from tag
        frame_remove_window(client->tag->frame, win);
        // and arrange monitor
        HSMonitor* m = find_monitor_with_tag(client->tag);
        if (m) monitor_apply_layout(m);
        // ignore events from it
        XSelectInput(g_display, win, 0);
        XUngrabButton(g_display, AnyButton, AnyModifier, win);
        // permanently remove it
        g_hash_table_remove(g_clients, &win);
    }

    // destroys a special client
    void destroy_client(HSClient* client) {
        g_free(client);
    }

    void window_unfocus(Window window) {
        // grab buttons in old window again
        XSetWindowBorder(g_display, window, wincolors[0][ColWindowBorder]);
        window_grab_button(window);
    }

    static Window lastfocus = 0;
    void window_unfocus_last() {
        if (lastfocus)    window_unfocus(lastfocus);

        // give focus to root window
        XSetInputFocus(g_display, g_root, RevertToPointerRoot, CurrentTime);
        if (lastfocus)
            ewmh_update_active_window(None);

        lastfocus = 0;
    }

    void window_focus(Window window) {
        // unfocus last one
        window_unfocus(lastfocus);
        // change window-colors
        XSetWindowBorder(g_display, window, wincolors[1][ColWindowBorder]);
        // set keyboardfocus
        XSetInputFocus(g_display, window, RevertToPointerRoot, CurrentTime);
        if (window != lastfocus) {
            /* FIXME: this is a workaround because window_focus always is called
             * twice.  see BUGS for more information
             *
             * only emit the hook if the focus *really* changes */
            ewmh_update_active_window(window);
        }
        lastfocus = window;
        /* do some specials for the max layout */
        bool is_max_layout = frame_focused_window(g_cur_frame) == window
                             && g_cur_frame->content.clients.layout == LAYOUT_MAX;
        if (is_max_layout)
          XRaiseWindow(g_display, window);
    }
```



```

    void client_setup_border(HSClient* client, bool focused) {
        XSetWindowBorder(g_display, client->window, wincolors[focused ? 1:0][ColWind
    owBorder]);
    }

    void client_resize(HSClient* client, XRectangle rect) {
        HSMonitor* m;
        if (client->fullscreen && (m = find_monitor_with_tag(client->tag))) {
            client_resize_fullscreen(client, m);
        } else if (client->pseudotile && (m = find_monitor_with_tag(client->tag))) {
            client_resize_floating(client, m);
        } else {
           // ensure minimum size
           if (rect.width < WINDOW_MIN_WIDTH)    rect.width = WINDOW_MIN_WIDTH;
           if (rect.height < WINDOW_MIN_HEIGHT)  rect.height = WINDOW_MIN_HEIGHT;

           if (!client) return;

           Window win = client->window;
           if (client) {
              if (RECTANGLE_EQUALS(client->last_size, rect)) return;
              client->last_size = rect;
           }
           // apply border width
           rect.width -= window_border_width * 2;
           rect.height -= window_border_width * 2;
           XSetWindowBorderWidth(g_display, win, window_border_width);
           XMoveResizeWindow(g_display, win, rect.x, rect.y, rect.width, rect.height
    );
        }
    }

    void client_resize_fullscreen(HSClient* client, HSMonitor* m) {
        if (!client || !m) return;

        XSetWindowBorderWidth(g_display, client->window, 0);
        client->last_size = m->rect;
        XMoveResizeWindow(g_display, client->window,
                          m->rect.x, m->rect.y, m->rect.width, m->rect.height);
    }

    void client_resize_floating(HSClient* client, HSMonitor* m) {
        if (!client || !m) return;
        if (client->fullscreen) {
            client_resize_fullscreen(client, m);
            return;
        }

        // ensure minimal size
        if (client->float_size.width < WINDOW_MIN_WIDTH)
            client->float_size.width = WINDOW_MIN_WIDTH;
        if (client->float_size.height < WINDOW_MIN_HEIGHT)
            client->float_size.height = WINDOW_MIN_HEIGHT;
        client->last_size = client->float_size;

        XRectangle rect = client->last_size;
        XSetWindowBorderWidth(g_display, client->window, window_border_width);
        XMoveResizeWindow(g_display, client->window,
            rect.x, rect.y, rect.width, rect.height);
    }

    void client_center(HSClient* client, HSMonitor *m) {
       if(!client || !m) return;
       if(client->fullscreen) {
          client_resize_fullscreen(client, m);
          return;
       }

       XRectangle size = client->float_size;
       size.x = m->rect.x + m->rect.width/2 - client->float_size.width/2;
       size.y = m->rect.y + m->rect.height/2 - client->float_size.height/2 + bh;
       client->float_size = size;
```

```
    client->last_size = size;
    XSetWindowBorderWidth(g_display, client->window, window_border_width);
    XMoveResizeWindow(g_display, client->window, size.x, size.y, size.width, size
.height);
}

XRectangle client_outer_floating_rect(HSClient* client) {
    XRectangle rect = client->float_size;
    rect.width  += window_border_width * 2;
    rect.height += window_border_width * 2;
    return rect;
}

void client_close(const Arg *arg) {
    XEvent ev;
    // if there is no focus, then there is nothing to do
    if (!g_cur_frame) return;
    Window win = frame_focused_window(g_cur_frame);
    if (!win) return;
    ev.type = ClientMessage;
    ev.xclient.window = win;
    ev.xclient.message_type = g_wmatom[WMProtocols];
    ev.xclient.format = 32;
    ev.xclient.data.l[0] = g_wmatom[WMDelete];
    ev.xclient.data.l[1] = CurrentTime;
    XSendEvent(g_display, win, False, NoEventMask, &ev);
}

void window_set_visible(Window win, bool visible) {
    XGrabServer(g_display);
    XSelectInput(g_display, win, CLIENT_EVENT_MASK & ~StructureNotifyMask);
    XSelectInput(g_display, g_root, ROOT_EVENT_MASK & ~SubstructureNotifyMask);
    if(visible)   XMapWindow(g_display, win);
    else          XUnmapWindow(g_display, win);
    XSelectInput(g_display, win, CLIENT_EVENT_MASK);
    XSelectInput(g_display, g_root, ROOT_EVENT_MASK);
    XUngrabServer(g_display);
}

void client_clear_urgent(HSClient* client) {
    if (client->urgent) {
        client->tag->urgent = false;
        client->urgent = false;
        XWMHints *wmh;
        if(!(wmh = XGetWMHints(g_display, client->window)))
            return;
        wmh->flags &= ~XUrgencyHint;
        XSetWMHints(g_display, client->window, wmh);
        XFree(wmh);
    }
}

void client_update_wm_hints(HSClient* client) {
    XWMHints* wmh = XGetWMHints(g_display, client->window);
    if (!wmh)   return;

    if ((frame_focused_window(g_cur_frame) == client->window)
        && wmh->flags & XUrgencyHint) {
        // remove urgency hint if window is focused
        wmh->flags &= ~XUrgencyHint;
        XSetWMHints(g_display, client->window, wmh);
    } else {
        bool newval = (wmh->flags & XUrgencyHint) ? true : false;
        if (newval != client->urgent) {
            client->urgent = newval;
            client->tag->urgent = client->urgent;
        }
    }
}

void client_update_title(HSClient* client) {
    gettextprop(client->window, g_netatom[NetWmName], client->title, sizeof(clie
```



```
nt->title));
    if(client->title[0] == '\0')
        strcpy(client->title, "broken");
}

HSClient* get_current_client() {
    Window win = frame_focused_window(g_cur_frame);
    if (!win) return NULL;
    return get_client_from_window(win);
}

void client_set_fullscreen(HSClient* client, bool state) {
    if (client->fullscreen == state)   return;

    client->fullscreen = state;
    if (state) {
        XChangeProperty(g_display, client->window, g_netatom[NetWmState], XA_ATOM
,
                32, PropModeReplace, (unsigned char *)&g_netatom[NetWmStateFullscre
en], 1);

         XRaiseWindow(g_display, client->window);
    } else {
        XChangeProperty(g_display, client->window, g_netatom[NetWmState], XA_ATOM
,
                32, PropModeReplace, (unsigned char *)0, 0);
    }

    monitor_apply_layout(find_monitor_with_tag(client->tag));
}

void client_set_pseudotile(HSClient* client, bool state) {
    client->pseudotile = state;
    HSFrame* f = frame_current_selection();
    size_t floatcount = f->content.clients.floatcount;
    floatcount += state ? 1 : -1;
    f->content.clients.floatcount = floatcount;
    if(state){
      client_center(client, find_monitor_with_tag(client->tag));
   }

    monitor_apply_layout(find_monitor_with_tag(client->tag));
}

void set_pseudotile(const Arg *arg){
   HSClient *client = get_current_client();
   if (!client) return;

   client_set_pseudotile(client, !client->pseudotile);
}

// rules applying //
void rules_apply(HSClient* client, int *manage) {
   const Rule *r;

   for(int i=0; i< LENGTH(custom_rules); i++){
      r = &custom_rules[i];
      bool rule_match = true; // if entire rule matches

      for(int j=0; j<LENGTH(g_condition_types); j++){
         if(!strcmp(g_condition_types[j].name, r->condition))
            rule_match = g_condition_types[j].matches(r->cond_value, client);
      }

      if (rule_match) {
         client->tag = find_tag(tags[r->tag]);
         *manage = (int)r->manage;
         client->pseudotile = r->pseudotile;
      }
   }
}

/// CONDITIONS ///
```

```
bool condition_class(char* cond_value, HSClient* client) {
    XClassHint hint;
    XGetClassHint(g_display, client->window, &hint);
    char* class_str = hint.res_class ? hint.res_class : "";
    bool match = !strcmp(cond_value, class_str);
    XFree(hint.res_name);
    XFree(hint.res_class);

    return match;
}

bool condition_title(char* cond_value, HSClient* client) {
    return !strcmp(cond_value, client->title);
}

bool condition_windowtype(char* cond_value, HSClient* client) {
    long bufsize = 10;
    char *buf;
    Atom type_ret, wintype;
    int format;
    unsigned long items, bytes_left;
    long offset = 0;

    int status = XGetWindowProperty( g_display, client->window, g_netatom[NetWmWindowType],
                                     offset, bufsize, False, ATOM("ATOM"),
                                     &type_ret, &format, &items, &bytes_left,
                                     (unsigned char**)&buf );
    // we only need precisely four bytes (one Atom)
    // if there are bytes left, something went wrong
    if(status != Success || bytes_left > 0 || items < 1 || buf == NULL) {
        return false;
    } else {
        wintype= *(Atom *)buf;
        XFree(buf);
    }

    for (int i = NetWmWindowTypeFIRST; i <= NetWmWindowTypeLAST; i++) {
        // try to find the window type
        if (wintype == g_netatom[i])
            return !strcmp(cond_value, g_netatom_names[i]);
    }

    return false;
}
```



```
/*
 * FusionWM - globals.c
 */

#include "globals.h"

#include <stdarg.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#include <X11/Xproto.h>
#include <X11/Xutil.h>

void die(const char *errstr, ...) {
    va_list ap;
    va_start(ap, errstr);
    vfprintf(stderr, errstr, ap);
    va_end(ap);
    exit(EXIT_FAILURE);
}

// get X11 color from color string
unsigned long getcolor(const char *colstr) {
    Colormap cmap = DefaultColormap(g_display, g_screen);
    XColor color;
    if(!XAllocNamedColor(g_display, cmap, colstr, &color, &color))
        die("error, cannot allocate color '%s'\n", colstr);
    return color.pixel;
}

bool gettextprop(Window w, Atom atom, char *text, unsigned int size) {
   char **list = NULL;
   int n;
   XTextProperty name;

   if(!text || size == 0) return False;

   text[0] = '\0';
   XGetTextProperty(g_display, w, &name, atom);
   if(!name.nitems) return False;

   if(name.encoding == XA_STRING) {
      strncpy(text, (char *)name.value, size - 1);
   } else if(XmbTextPropertyToTextList(g_display, &name, &list, &n) >= Success && n > 0 && *list) {
         strncpy(text, *list, size - 1);
         XFreeStringList(list);
   }
   text[size - 1] = '\0';
   XFree(name.value);
   return True;
}

void spawn(const Arg *arg){
   char *sh = NULL;
   pid_t pid;

   if(!(sh = getenv("SHELL"))) sh = "/bin/sh";

   if((pid = fork()) == 0){
      if(g_display) close(ConnectionNumber(g_display));

      setsid();
      execl(sh, sh, "-c", (char*)arg->v, (char*)NULL);
   }
}

void quit(const Arg *arg) {
   g_aboutToQuit = true;
}
```

```c
/*
 * FusionWM - inputs.c
 */

#include "globals.h"
#include "inputs.h"
#include "layout.h"
#include "clientlist.h"
#include "config.h"

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include <X11/XKBlib.h>
#include <X11/cursorfont.h>

#include "binds.h"

static unsigned int numlockmask = 0;

static XButtonPressedEvent g_button_drag_start;
static XRectangle       g_win_drag_start;
static HSClient*        g_win_drag_client = NULL;
static HSMonitor*       g_drag_monitor = NULL;
static MouseBinding*    g_drag_bind = NULL;

static Cursor g_cursor;
static unsigned int numlockmask;

#define REMOVEBUTTONMASK(mask) ((mask) & ~( Button1Mask|Button2Mask|Button3Mask|Button4Mask|Button5Mask ))

void mouse_start_drag(XEvent* ev) {
    XButtonEvent* be = &(ev->xbutton);
    g_drag_bind = mouse_binding_find(be->state, be->button);
    if (!g_drag_bind) {
        // there is no valid bind for this type of mouse event
        return;
    }
    Window win = ev->xbutton.subwindow;
    g_win_drag_client = get_client_from_window(win);
    if (!g_win_drag_client) {
        g_drag_bind = NULL;
        return;
    }
    if (!g_win_drag_client->pseudotile) {
        // only can drag wins in  floating mode or pseudotile
        g_win_drag_client = NULL;
        g_drag_bind = NULL;
        return;
    }
    g_win_drag_start = g_win_drag_client->float_size;
    g_button_drag_start = ev->xbutton;
    g_drag_monitor = get_current_monitor();
    XGrabPointer(g_display, win, True,
        PointerMotionMask|ButtonReleaseMask, GrabModeAsync,
            GrabModeAsync, None, None, CurrentTime);
}

void mouse_stop_drag() {
    g_win_drag_client = NULL;
    g_drag_bind = NULL;
    XUngrabPointer(g_display, CurrentTime);
}

void handle_motion_event(XEvent* ev) {
    if (g_drag_monitor != get_current_monitor()) {
        mouse_stop_drag();
        return;
    }
    if (!g_win_drag_client) return;
    if (!g_drag_bind) return;
    if (ev->type != MotionNotify) return;
    MouseFunction function = g_drag_bind->function;
    if (!function) return;
    // call function that handles it
    function(&(ev->xmotion));
}

static void grab_button(MouseBinding* mb) {
    unsigned int modifiers[] = { 0, LockMask, numlockmask, numlockmask|LockMask };
    // grab button for each modifier that is ignored (capslock, numlock)
    for (int i = 0; i < LENGTH(modifiers); i++) {
        XGrabButton(g_display, mb->button, modifiers[i]|mb->mask,
                    g_root, True, ButtonPressMask,
                    GrabModeAsync, GrabModeAsync, None, None);
    }
}

void grab_buttons() {
   update_numlockmask();
   // init modifiers after updating numlockmask
   XUngrabButton(g_display, AnyButton, AnyModifier, g_root);
   for(int i=0; i<2; i++) grab_button(&buttons[i]);
}

MouseBinding* mouse_binding_find(unsigned int modifiers, unsigned int button) {
   for(int i=0; i<LENGTH(buttons); i++){
      MouseBinding * mb = &buttons[i];
      if((REMOVEBUTTONMASK(CLEANMASK(modifiers)) == REMOVEBUTTONMASK(CLEANMASK(mb->mask))) &&
         (button == mb->button))
         return mb;
   }
   return NULL;
}

void mouse_function_move(XMotionEvent* me) {
    int x_diff = me->x_root - g_button_drag_start.x_root;
    int y_diff = me->y_root - g_button_drag_start.y_root;
    g_win_drag_client->float_size = g_win_drag_start;
    g_win_drag_client->float_size.x += x_diff;
    g_win_drag_client->float_size.y += y_diff;
    // snap it to other windows
    int dx, dy;
    client_snap_vector(g_win_drag_client, g_win_drag_client->tag,
                       SNAP_EDGE_ALL, &dx, &dy);
    g_win_drag_client->float_size.x += dx;
    g_win_drag_client->float_size.y += dy;
    client_resize_floating(g_win_drag_client, g_drag_monitor);
}

void mouse_function_resize(XMotionEvent* me) {
    int x_diff = me->x_root - g_button_drag_start.x_root;
    int y_diff = me->y_root - g_button_drag_start.y_root;
    g_win_drag_client->float_size = g_win_drag_start;
    // relative x/y coords in drag window
    int rel_x = g_button_drag_start.x_root - g_win_drag_start.x;
    int rel_y = g_button_drag_start.y_root - g_win_drag_start.y;
    bool top = false;
    bool left = false;
    if (rel_y < g_win_drag_start.height/2) {
        top = true;
        y_diff *= -1;
    }
    if (rel_x < g_win_drag_start.width/2) {
        left = true;
        x_diff *= -1;
    }
    // avoid an overflow
    int new_width  = g_win_drag_client->float_size.width + x_diff;
    int new_height = g_win_drag_client->float_size.height + y_diff;
    if (left)   g_win_drag_client->float_size.x -= x_diff;
    if (top)    g_win_drag_client->float_size.y -= y_diff;
```

```
    if (new_width <  WINDOW_MIN_WIDTH)  new_width = WINDOW_MIN_WIDTH;
    if (new_height < WINDOW_MIN_HEIGHT) new_height = WINDOW_MIN_HEIGHT;
    g_win_drag_client->float_size.width  = new_width;
    g_win_drag_client->float_size.height = new_height;
    // snap it to other windows
    int dx, dy;
    int snap_flags = 0;
    if (left)   snap_flags |= SNAP_EDGE_LEFT;
    else        snap_flags |= SNAP_EDGE_RIGHT;
    if (top)    snap_flags |= SNAP_EDGE_TOP;
    else        snap_flags |= SNAP_EDGE_BOTTOM;
    client_snap_vector(g_win_drag_client, g_win_drag_client->tag,
                       snap_flags, &dx, &dy);
    if (left) {
        g_win_drag_client->float_size.x += dx;
        dx *= -1;
    }
    if (top) {
        g_win_drag_client->float_size.y += dy;
        dy *= -1;
    }
    g_win_drag_client->float_size.width += dx;
    g_win_drag_client->float_size.height += dy;
    client_resize_floating(g_win_drag_client, g_drag_monitor);
}

struct SnapData {
    HSClient*       client;
    XRectangle      rect;
    enum SnapFlags  flags;
    int             dx, dy; // the vector from client to other to make them snap
};

static bool is_point_between(int point, int left, int right) {
    return (point < right && point >= left);
}

static bool intervals_intersect(int a_left, int a_right, int b_left, int b_right
) {
    return is_point_between(a_left, b_left, b_right)
        || is_point_between(a_right, b_left, b_right)
        || is_point_between(b_right, a_left, a_right)
        || is_point_between(b_left, a_left, a_right);
}

// compute vector to snap a point to an edge
static void snap_1d(int x, int edge, int* delta) {
    // whats the vector from subject to edge?
    int cur_delta = edge - x;
    // if distance is smaller then all other deltas
    if (abs(cur_delta) < abs(*delta)) {
        // then snap it, i.e. save vector
        *delta = cur_delta;
    }
}

static int client_snap_helper(HSClient* candidate, struct SnapData* d) {
   if (candidate == d->client)  return 0;

   XRectangle subject  = d->rect;
   XRectangle other    = client_outer_floating_rect(candidate);
   if (intervals_intersect(other.y, other.y + other.height, subject.y, subject.y
 + subject.height)) {
      // check if x can snap to the right
      if (d->flags & SNAP_EDGE_RIGHT)  snap_1d(subject.x + subject.width, other.
x, &d->dx);
      // or to the left
      if (d->flags & SNAP_EDGE_LEFT)   snap_1d(subject.x, other.x + other.width,
 &d->dx);
   }
   if (intervals_intersect(other.x, other.x + other.width, subject.x, subject.x
+ subject.width)) {
      // if we can snap to the top
```



```
      if (d->flags & SNAP_EDGE_TOP)    snap_1d(subject.y, other.y + other.height
, &d->dy);
      // or to the bottom
      if (d->flags & SNAP_EDGE_BOTTOM) snap_1d(subject.y + subject.height, other
.y, &d->dy);
   }
   return 0;
}

// get the vector to snap a client to it's neighbour
void client_snap_vector(struct HSClient* client, struct HSTag* tag,
                        enum SnapFlags flags,
                        int* return_dx, int* return_dy) {
    struct SnapData d;
    int distance = (snap_distance > 0) ? snap_distance : 0;
    // init delta
    *return_dx = 0;
    *return_dy = 0;
    if (!distance)  return;

    d.client    = client;
    d.rect      = client_outer_floating_rect(client);
    d.flags     = flags;
    d.dx        = distance;
    d.dy        = distance;

    // snap to monitor edges
    HSMonitor* m = g_drag_monitor;
    if (flags & SNAP_EDGE_TOP)      snap_1d(d.rect.y, 0, &d.dy);
    if (flags & SNAP_EDGE_LEFT)     snap_1d(d.rect.x, 0, &d.dx);
    if (flags & SNAP_EDGE_RIGHT)    snap_1d(d.rect.x + d.rect.width, m->rect.wid
th, &d.dx);
    if (flags & SNAP_EDGE_BOTTOM)   snap_1d(d.rect.y + d.rect.height, m->rect.he
ight - bh, &d.dy);

    // snap to other clients
    frame_foreach_client(tag->frame, (ClientAction)client_snap_helper, &d);

    // write back results
    if (abs(d.dx) < abs(distance))  *return_dx = d.dx;
    if (abs(d.dy) < abs(distance))  *return_dy = d.dy;
}

//--------
void inputs_init() {
   grab_keys();
   grab_buttons();

   /* set cursor theme */
   g_cursor = XCreateFontCursor(g_display, XC_left_ptr);
   XDefineCursor(g_display, g_root, g_cursor);
}

void inputs_destroy() {
    XFreeCursor(g_display, g_cursor);
}

void key_press(XEvent* ev) {
    unsigned int i;
    KeySym keysym;
    XKeyEvent *event;

    event = &ev->xkey;
    keysym = XkbKeycodeToKeysym(g_display, (KeyCode)event->keycode, 0, 0);
    for(i =0; i<LENGTH(keys); i++)
       if(keysym == keys[i].keysym &&
          CLEANMASK(keys[i].mod) == CLEANMASK(event->state) &&
          keys[i].func)
          keys[i].func(&(keys[i].arg));
}

void grab_keys(void) {
   update_numlockmask();
```

```
   unsigned int i, j;
   unsigned int modifiers[] = { 0, LockMask, numlockmask, numlockmask|LockMask }
;
   KeyCode code;

   XUngrabKey(g_display, AnyKey, AnyModifier, g_root); //remove all current grab
s
   for(i = 0; i < LENGTH(keys); i++)
      if((code = XKeysymToKeycode(g_display, keys[i].keysym)))
         for(j = 0; j < LENGTH(modifiers); j++)
            XGrabKey(g_display, code, keys[i].mod | modifiers[j], g_root,
                  True, GrabModeAsync, GrabModeAsync);
}

// update the numlockmask
void update_numlockmask() {
    unsigned int i, j;
    XModifierKeymap *modmap;

    numlockmask = 0;
    modmap = XGetModifierMapping(g_display);
    for(i = 0; i < 8; i++)
        for(j = 0; j < modmap->max_keypermod; j++)
            if(modmap->modifiermap[i * modmap->max_keypermod + j]
               == XKeysymToKeycode(g_display, XK_Num_Lock))
                numlockmask = (1 << i);
    XFreeModifiermap(modmap);
}
```



```
/*
 * FusionWM - layout.c
 */

#include "clientlist.h"
#include "globals.h"
#include "layout.h"
#include "config.h"

#include <stdio.h>
#include <string.h>
#include <assert.h>
#ifdef XINERAMA
   #include <X11/extensions/Xinerama.h>
#endif //XINERAMA //

// status
static char stext[256];

typedef struct {
   int x, y, w, h;
   unsigned long colors[NUMCOLORS][ColLast];
   Drawable drawable;
   GC gc;
   struct {
      int ascent;
      int descent;
      int height;
      XFontSet set;
      XFontStruct *xfont;
   } font;
} DC; // draw context
static DC dc;

char* g_layout_names[] = {
    "vertical",
    "horizontal",
    "max",
    NULL,
};

void layout_init() {
    g_cur_monitor = 0;
    g_tags = g_array_new(false, false, sizeof(HSTag*));
    g_monitors = g_array_new(false, false, sizeof(HSMonitor));

    // init font
    initfont(font);

    //init colors
    for(int i=0; i<NUMCOLORS; i++){
      dc.colors[i][ColFrameBorder] = getcolor(colors[i][ColFrameBorder]);
      dc.colors[i][ColWindowBorder] = getcolor(colors[i][ColWindowBorder]);
      dc.colors[i][ColFG] = getcolor(colors[i][ColFG]);
      dc.colors[i][ColBG] = getcolor(colors[i][ColBG]);
    }
    dc.drawable = XCreatePixmap(g_display, g_root,
                                DisplayWidth(g_display, DefaultScreen(g_display
)),
                                bh, DefaultDepth(g_display, DefaultScreen(g_dis
play)));
    dc.gc = XCreateGC(g_display, g_root, 0, NULL);
    dc.h = bh;

    if(!dc.font.set) XSetFont(g_display, dc.gc, dc.font.xfont->fid);

    for(int i=0; i<LENGTH(tags); i++)
       add_tag(tags[i]);

    monitors_init();
}

void layout_destroy() {
```

```
    int i;
    for (i = 0; i < g_tags->len; i++) {
        HSTag* tag = g_array_index(g_tags, HSTag*, i);
        frame_do_recursive(tag->frame, frame_show_clients, 2);
        g_string_free(tag->name, true);
        g_free(tag);
    }
    g_array_free(g_tags, true);
    g_array_free(g_monitors, true);
}

HSFrame* frame_create_empty() {
    HSFrame* frame = g_new0(HSFrame, 1);
    frame->type = TYPE_CLIENTS;
    frame->window_visible = false;
    frame->content.clients.layout = default_frame_layout;
    // set window atributes
    XSetWindowAttributes at;
    at.background_pixmap = ParentRelative;
    at.override_redirect = True;
    at.bit_gravity       = StaticGravity;
    at.event_mask        = SubstructureRedirectMask|SubstructureNotifyMask
         |ExposureMask|VisibilityChangeMask
         |EnterWindowMask|LeaveWindowMask|FocusChangeMask;
    frame->window = XCreateWindow(g_display, g_root,
                        42, 42, 42, 42, frame_border_width,
                        DefaultDepth(g_display, DefaultScreen(g_display)),
                        CopyFromParent,
                        DefaultVisual(g_display, DefaultScreen(g_display)),
                        CWOverrideRedirect|CWBackPixmap|CWEventMask, &at);
    return frame;
}

void frame_insert_window(HSFrame* frame, Window window) {
    if (frame->type == TYPE_CLIENTS) {
        // insert it here
        Window* buf = frame->content.clients.buf;
        size_t count = frame->content.clients.count;
        count++;

        HSClient *c = get_client_from_window(window);
        size_t floatcount = frame->content.clients.floatcount;
        if(c->pseudotile)
           floatcount++;

        // insert it after the selection
        int index = frame->content.clients.selection + 1;
        index = CLAMP(index, 0, count - 1);
        buf = g_renew(Window, buf, count);
        // shift other windows to the back to insert the new one at index
        memmove(buf + index + 1, buf + index, sizeof(*buf) * (count - index - 1)
);
        buf[index] = window;
        // write results back
        frame->content.clients.count = count;
        frame->content.clients.buf = buf;
        frame->content.clients.floatcount = floatcount;
        // check for focus
        if ((g_cur_frame == frame && frame->content.clients.selection >= (count-
1))
           || focus_new_clients ) {
            frame->content.clients.selection = count - 1;
            window_focus(window);
        }
    } else { /* frame->type == TYPE_FRAMES */
        HSLayout* layout = &frame->content.layout;
        frame_insert_window((layout->selection == 0)? layout->a : layout->b, win
dow);
    }
}

bool frame_remove_window(HSFrame* frame, Window window) {
    if (frame->type == TYPE_CLIENTS) {
```



```
        Window* buf = frame->content.clients.buf;
        HSClient *c = get_client_from_window(window);
        size_t count = frame->content.clients.count;
        size_t floatcount = frame->content.clients.floatcount;
        int i;
        for (i = 0; i < count; i++) {
            if (buf[i] == window) {
                // if window was found, then remove it
                memmove(buf+i, buf+i+1, sizeof(Window)*(count - i - 1));
                count--;
                if(c->pseudotile)   floatcount--;
                buf = g_renew(Window, buf, count);
                frame->content.clients.buf = buf;
                frame->content.clients.count = count;
                frame->content.clients.floatcount = floatcount;
                // find out new selection
                int selection = frame->content.clients.selection;
                // if selection was before removed window
                // then do nothing, else shift it by 1
                selection -= (selection < i) ? 0 : 1;
                // ensure, that it's a valid index
                selection = count ? CLAMP(selection, 0, count-1) : 0;
                frame->content.clients.selection = selection;
                return true;
            }
        }
        return false;
    } else { /* frame->type == TYPE_FRAMES */
        bool found = frame_remove_window(frame->content.layout.a, window);
        found = found || frame_remove_window(frame->content.layout.b, window);
        return found;
    }
}

void frame_destroy(HSFrame* frame, Window** buf, size_t* count) {
    if (frame->type == TYPE_CLIENTS) {
        *buf = frame->content.clients.buf;
        *count = frame->content.clients.count;
    } else { /* frame->type == TYPE_FRAMES */
        size_t c1, c2;
        Window *buf1, *buf2;
        frame_destroy(frame->content.layout.a, &buf1, &c1);
        frame_destroy(frame->content.layout.b, &buf2, &c2);
        // append buf2 to buf1
        buf1 = g_renew(Window, buf1, c1 + c2);
        memcpy(buf1+c1, buf2, sizeof(Window) * c2);
        // free unused things
        g_free(buf2);
        // return;
        *buf = buf1;
        *count = c1 + c2;
    }
    // free other things
    XDestroyWindow(g_display, frame->window);
    g_free(frame);
}

int find_layout_by_name(char* name) {
    for (int i = 0; i < LENGTH(g_layout_names); i++) {
        if (!g_layout_names[i])  break;

        if (!strcmp(name, g_layout_names[i]))
            return i;
    }
    return -1;
}

void monitor_apply_layout(HSMonitor* monitor) {
    if (monitor) {
        XRectangle rect = monitor->rect;
        // apply pad
        rect.y += bh;
        rect.height -= bh;
```

```
215         // apply window gap
            rect.x += window_gap;
            rect.y += window_gap;
            rect.height -= window_gap;
            rect.width -= window_gap;
220         frame_apply_layout(monitor->tag->frame, rect);

            if (get_current_monitor() == monitor)
                frame_focus_recursive(monitor->tag->frame);

225         draw_bar(monitor);
        }
    }

    void set_layout(const Arg *arg) {
230     int layout = 0;

        layout = find_layout_by_name((char*)arg->v);
        if (layout < 0) return;

235     if (g_cur_frame && g_cur_frame->type == TYPE_CLIENTS) {
            g_cur_frame->content.clients.layout = layout;
            monitor_apply_layout(get_current_monitor());
        }
        return;
240 }

    void frame_apply_client_layout(HSFrame* frame, XRectangle rect, int layout) {
        Window* buf = frame->content.clients.buf;
        size_t count = frame->content.clients.count;
245     size_t count_wo_floats = count - frame->content.clients.floatcount;
        int selection = frame->content.clients.selection;
        XRectangle cur = rect;
        int last_step_y;
        int last_step_x;
250     int step_y;
        int step_x;

       if(layout == LAYOUT_MAX){
            for (int i = 0; i < count; i++) {
255            HSClient* client = get_client_from_window(buf[i]);
               client_setup_border(client, (g_cur_frame == frame) && (i == selection)
    );
               client_resize(client, rect);
               if (i == selection)
                  XRaiseWindow(g_display, buf[i]);
260         }
            return;
        }
        if(count_wo_floats > 0) {
           if (layout == LAYOUT_VERTICAL) {
265           // only do steps in y direction
              last_step_y = cur.height % count_wo_floats; // get the space on bottom
              last_step_x = 0;
              cur.height /= count_wo_floats;
              step_y = cur.height;
270           step_x = 0;
           } else {
              // only do steps in x direction
              last_step_y = 0;
              last_step_x = cur.width % count_wo_floats; // get the space on the rig
    ht
275           cur.width /= count_wo_floats;
              step_y = 0;
              step_x = cur.width;
           }
           for (int i = 0; i < count; i++) {
280           HSClient* client = get_client_from_window(buf[i]);
              if(client->pseudotile) continue;
              // add the space, if count doesnot divide frameheight without remainde
    r
              cur.height += (i == count-1) ? last_step_y : 0;
              cur.width += (i == count-1) ? last_step_x : 0;
```



```
285         client_setup_border(client, (g_cur_frame == frame) && (i == selection)
    );
            client_resize(client, cur);
            cur.y += step_y;
            cur.x += step_x;
          }
290     }
    }

    void frame_apply_layout(HSFrame* frame, XRectangle rect) {
        if (frame->type == TYPE_CLIENTS) {
295         size_t count = frame->content.clients.count;
            // frame only -> apply window_gap
            rect.height -= window_gap;
            rect.width -= window_gap;
            // apply frame width
300         rect.x += frame_border_width;
            rect.y += frame_border_width;
            rect.height -= frame_border_width * 2;
            rect.width -= frame_border_width * 2;
            if (rect.width <= WINDOW_MIN_WIDTH || rect.height <= WINDOW_MIN_HEIGHT)
    {
305             // do nothing on invalid size
                return;
            }
            XSetWindowBorderWidth(g_display, frame->window, frame_border_width);
            // set indicator frame
310         unsigned long border_color = dc.colors[0][ColFrameBorder];
            if (g_cur_frame == frame)
                border_color = dc.colors[1][ColFrameBorder];

            XSetWindowBorder(g_display, frame->window, border_color);
315         XMoveResizeWindow(g_display, frame->window,
                              rect.x - frame_border_width,
                              rect.y - frame_border_width,
                              rect.width, rect.height);
            XSetWindowBackgroundPixmap(g_display, frame->window, ParentRelative);
320         XClearWindow(g_display, frame->window);
            XLowerWindow(g_display, frame->window);
            frame_set_visible(frame, (count != 0) || (g_cur_frame == frame));
            // move windows
            if (count == 0) return;
325
            frame_apply_client_layout(frame, rect, frame->content.clients.layout);
        } else { /* frame->type == TYPE_FRAMES */
            HSLayout* layout = &frame->content.layout;
            XRectangle first = rect;
330         XRectangle second = rect;
            if (layout->align == ALIGN_VERTICAL) {
                first.height = (rect.height * layout->fraction) / FRACTION_UNIT;
                second.y += first.height;
                second.height -= first.height;
335         } else { // (layout->align == ALIGN_HORIZONTAL)
                first.width = (rect.width * layout->fraction) / FRACTION_UNIT;
                second.x += first.width;
                second.width -= first.width;
            }
340         frame_set_visible(frame, false);
            frame_apply_layout(layout->a, first);
            frame_apply_layout(layout->b, second);
        }
    }
345
    void add_monitor(XRectangle rect, HSTag* tag, int primary) {
        assert(tag != NULL);
        HSMonitor m;
        memset(&m, 0, sizeof(m));
350     m.rect = rect;
        m.tag = tag;
        m.mouse.x = 0;
        m.mouse.y = 0;
        m.primary = primary;
355     create_bar(&m);
```

```
    g_array_append_val(g_monitors, m);
}

HSMonitor* find_monitor_with_tag(HSTag* tag) {
    int i;
    for (i = 0; i < g_monitors->len; i++) {
        HSMonitor* m = &g_array_index(g_monitors, HSMonitor, i);
        if (m->tag == tag)
            return m;
    }
    return NULL;
}

HSTag* find_tag(const char* name) {
    int i;
    for (i = 0; i < g_tags->len; i++) {
        if (!strcmp(g_array_index(g_tags, HSTag*, i)->name->str, name))
            return g_array_index(g_tags, HSTag*, i);
    }
    return NULL;
}

void add_tag(const char* name) {
    HSTag* find_result = find_tag(name);
    if (find_result)  return;

    HSTag* tag = g_new(HSTag, 1);
    tag->frame = frame_create_empty();
    tag->name = g_string_new(name);
    tag->urgent = false;
    g_array_append_val(g_tags, tag);
}

#ifdef XINERAMA
Bool is_unique_geometry(XineramaScreenInfo *unique, size_t n, XineramaScreenInfo
 *info) {
   while(n--)
      if(unique[n].x_org == info->x_org && unique[n].y_org == info->y_org
         && unique[n].width == info->width && unique[n].height == info->height)
         return False;

   return True;
}
#endif

void monitors_init() {
#ifdef XINERAMA
   if(XineramaIsActive(g_display)){
      int i, j, nn;
      XineramaScreenInfo *info = XineramaQueryScreens(g_display, &nn);
      XineramaScreenInfo *unique = NULL;

      if(!(unique = (XineramaScreenInfo *)malloc(sizeof(XineramaScreenInfo)*nn))
)
         die("fatal: could not malloc() %u bytes\n", sizeof(XineramaScreenInfo)*nn);
      for(i=0, j=0; i<nn; i++)
         if(is_unique_geometry(unique, j, &info[i]))
            memcpy(&unique[j++], &info[i], sizeof(XineramaScreenInfo));
      XFree(info);
      nn = j;

      for(i=0; i<nn; i++){
         XRectangle rect = {
            .x = unique[i].x_org,
            .y = unique[i].y_org,
            .width =  unique[i].width,
            .height = unique[i].height,
         };
         HSTag *cur_tag = g_array_index(g_tags, HSTag*, i);
         if(i==0){ // first one is primary monitor
            add_monitor(rect, cur_tag, 1);
            g_cur_monitor = 0;
            g_cur_frame = cur_tag->frame;
```



```
         } else {
            add_monitor(rect, cur_tag, 0);
         }
      }
   } else
#endif //XINERAMA //
   {  // Default monitor setup
      XRectangle rect = {
         .x = 0,
         .y = 0,
         .width = DisplayWidth(g_display, DefaultScreen(g_display)),
         .height = DisplayHeight(g_display, DefaultScreen(g_display)),
      };
      // add monitor with first tag
      HSTag *cur_tag = g_array_index(g_tags, HSTag*, 0);
      add_monitor(rect, cur_tag, 1);
      g_cur_monitor = 0;
      g_cur_frame = cur_tag->frame;
   }
}

HSFrame* frame_current_selection() {
    HSMonitor* m = get_current_monitor();
    if (!m->tag) return NULL;
    HSFrame* frame = m->tag->frame;
    while (frame->type == TYPE_FRAMES) {
        frame = (frame->content.layout.selection == 0) ?
                frame->content.layout.a :
                frame->content.layout.b;
    }
    return frame;
}

void cycle(const Arg *arg) {
    // find current selection
    HSFrame* frame = frame_current_selection();
    if (frame->content.clients.count == 0)  return;

    int index = frame->content.clients.selection;
    int count = (int) frame->content.clients.count;
    if(index == count-1) index = 0;
    else                 index++;
    frame->content.clients.selection = index;
    Window window = frame->content.clients.buf[index];
    window_focus(window);
    XRaiseWindow(g_display, window);
}

void frame_split(HSFrame* frame, int align, int fraction) {
    // ensure fraction is allowed
    fraction = CLAMP(fraction,
                     FRACTION_UNIT * (0.0 + FRAME_MIN_FRACTION),
                     FRACTION_UNIT * (1.0 - FRAME_MIN_FRACTION));

    HSFrame* first = frame_create_empty();
    first->content = frame->content;
    first->type = TYPE_CLIENTS;
    first->parent = frame;

    HSFrame* second = frame_create_empty();
    second->type = TYPE_CLIENTS;
    second->parent = frame;

    frame->type = TYPE_FRAMES;
    frame->content.layout.align = align;
    frame->content.layout.a = first;
    frame->content.layout.b = second;
    frame->content.layout.selection = 0;
    frame->content.layout.fraction = fraction;
    // set focus
    g_cur_frame = first;
    // redraw monitor if exists
    monitor_apply_layout(monitor_with_frame(frame));
```

```
}

void split_v(const Arg *arg){
   int fraction = FRACTION_UNIT* CLAMP(arg->f,
                                       0.0 + FRAME_MIN_FRACTION,
                                       1.0 - FRAME_MIN_FRACTION);
   HSFrame *frame = frame_current_selection();
   frame_split(frame, ALIGN_VERTICAL, fraction);
}

void split_h(const Arg *arg){
   int fraction = FRACTION_UNIT* CLAMP(arg->f,
                                       0.0 + FRAME_MIN_FRACTION,
                                       1.0 - FRAME_MIN_FRACTION);
   HSFrame *frame = frame_current_selection();
   frame_split(frame, ALIGN_HORIZONTAL, fraction);
}

void resize_frame(const Arg *arg){
    char direction = ((char*)arg->v)[0];
    int delta = FRACTION_UNIT * resize_step;

    // if direction is left or up we have to flip delta because e.g. resize up
    // by 0.1 actually means: reduce fraction by 0.1, i.e. delta = -0.1
    switch (direction) {
        case 'l':   delta *= -1; break;
        case 'r':   break;
        case 'u':   delta *= -1; break;
        case 'd':   break;
        default:    return;
    }
    HSFrame* neighbour = frame_neighbour(g_cur_frame, direction);
    if (!neighbour) {
        // then try opposite direction
        switch (direction) {
            case 'l':   direction = 'r'; break;
            case 'r':   direction = 'l'; break;
            case 'u':   direction = 'd'; break;
            case 'd':   direction = 'u'; break;
            default:    assert(false); break;
        }
        neighbour = frame_neighbour(g_cur_frame, direction);
        if (!neighbour)  return ;
    }
    HSFrame* parent = neighbour->parent;
    assert(parent != NULL); // if has neighbour, it also must have a parent
    assert(parent->type == TYPE_FRAMES);
    int fraction = parent->content.layout.fraction;
    fraction += delta;
    fraction = CLAMP(fraction,
                     (int)(FRAME_MIN_FRACTION * FRACTION_UNIT),
                     (int)((1.0 - FRAME_MIN_FRACTION) * FRACTION_UNIT));
    parent->content.layout.fraction = fraction;
    // arrange monitor
    monitor_apply_layout(get_current_monitor());
}

HSMonitor* monitor_with_frame(HSFrame* frame) {
    // find toplevel Frame
    while (frame->parent)
        frame = frame->parent;

    HSTag* tag;
    for(int i=0; i<g_tags->len; i++){
      tag = g_array_index(g_tags, HSTag*, i);
      if(tag->frame == frame)   break;
    }
    return find_monitor_with_tag(tag);
}

HSFrame* frame_neighbour(HSFrame* frame, char direction) {
    HSFrame* other;
    bool found = false;
```



```
    while (frame->parent) {
        // find frame, where we can change the
        // selection in the desired direction
        HSLayout* layout = &frame->parent->content.layout;
        switch(direction) {
            case 'r':
                if (layout->align == ALIGN_HORIZONTAL && layout->a == frame) {
                    found = true;
                    other = layout->b;
                }
                break;
            case 'l':
                if (layout->align == ALIGN_HORIZONTAL && layout->b == frame) {
                    found = true;
                    other = layout->a;
                }
                break;
            case 'd':
                if (layout->align == ALIGN_VERTICAL && layout->a == frame) {
                    found = true;
                    other = layout->b;
                }
                break;
            case 'u':
                if (layout->align == ALIGN_VERTICAL && layout->b == frame) {
                    found = true;
                    other = layout->a;
                }
                break;
            default:
                return NULL;
                break;
        }
        if (found)   break;
        // else: go one step closer to root
        frame = frame->parent;
    }
    if (!found)   return NULL;

    return other;
}

// finds a neighbour within frame in the specified direction
// returns its index or -1 if there is none
int frame_inner_neighbour_index(HSFrame* frame, char direction) {
    int index = -1;
    if (frame->type != TYPE_CLIENTS) {
        fprintf(stderr, "warning: frame has invalid type\n");
        return -1;
    }
    int selection = frame->content.clients.selection;
    int count = frame->content.clients.count;
    switch (frame->content.clients.layout) {
        case LAYOUT_VERTICAL:
            if (direction == 'd') index = selection + 1;
            if (direction == 'u') index = selection - 1;
            break;
        case LAYOUT_HORIZONTAL:
            if (direction == 'r') index = selection + 1;
            if (direction == 'l') index = selection - 1;
            break;
        case LAYOUT_MAX:
            break;
        default:
            break;
    }
    // check that index is valid
    if (index < 0 || index >= count) {
        index = -1;
    }
    return index;
}
```

```
    void focus(const Arg *arg){
       char direction = ((char*)arg->v)[0];
       int index;

650    if ((index = frame_inner_neighbour_index(g_cur_frame, direction)) != -1) {
          g_cur_frame->content.clients.selection = index;
          frame_focus_recursive(g_cur_frame);
          monitor_apply_layout(get_current_monitor());
       } else {
655       HSFrame* neighbour = frame_neighbour(g_cur_frame, direction);
          if (neighbour != NULL) { // if neighbour was found
             HSFrame* parent = neighbour->parent;
             // alter focus (from 0 to 1, from 1 to 0)
             int selection = parent->content.layout.selection;
660          selection = (selection == 1) ? 0 : 1;
             parent->content.layout.selection = selection;
             // change focus if possible
             frame_focus_recursive(parent);
             monitor_apply_layout(get_current_monitor());
665       }
       }
    }

    void shift(const Arg *arg) {
670     char direction = ((char *)arg->v)[0];
        int index;

        if ((index = frame_inner_neighbour_index(g_cur_frame, direction)) != -1) {
            int selection = g_cur_frame->content.clients.selection;
675         Window* buf = g_cur_frame->content.clients.buf;
            // if internal neighbour was found, then swap
            Window tmp = buf[selection];
            buf[selection] = buf[index];
            buf[index] = tmp;
680
            if (focus_follows_shift) {
                g_cur_frame->content.clients.selection = index;
            }
            frame_focus_recursive(g_cur_frame);
685         monitor_apply_layout(get_current_monitor());
        } else {
            HSFrame* neighbour = frame_neighbour(g_cur_frame, direction);
            Window win = frame_focused_window(g_cur_frame);
            if (win && neighbour != NULL) { // if neighbour was found
690             // move window to neighbour
                frame_remove_window(g_cur_frame, win);
                frame_insert_window(neighbour, win);
                if (focus_follows_shift) {
                    // change selection in parrent
695                 HSFrame* parent = neighbour->parent;
                    assert(parent);
                    parent->content.layout.selection = ! parent->content.layout.sele
    ction;
                    frame_focus_recursive(parent);
                    // focus right window in frame
700                 HSFrame* frame = g_cur_frame;
                    assert(frame);
                    int i;
                    Window* buf = frame->content.clients.buf;
                    size_t count = frame->content.clients.count;
705                 for (i = 0; i < count; i++) {
                        if (buf[i] == win) {
                            frame->content.clients.selection = i;
                            window_focus(buf[i]);
                            break;
710                     }
                    }
                } else {
                    frame_focus_recursive(g_cur_frame);
                }
715             // layout was changed, so update it
                monitor_apply_layout(get_current_monitor());
            }
```



```
        }
    }
720
    Window frame_focused_window(HSFrame* frame) {
        if (!frame)
            return (Window)0;

725     // follow the selection to a leave
        while (frame->type == TYPE_FRAMES) {
            frame = (frame->content.layout.selection == 0) ?
                    frame->content.layout.a : frame->content.layout.b;
        }
730     if (frame->content.clients.count) {
            int selection = frame->content.clients.selection;
            return frame->content.clients.buf[selection];
        } // else, if there are no windows
        return (Window)0;
735 }

    // try to focus window in frame
    // returns true if win was found and focused, else returns false
    bool frame_focus_window(HSFrame* frame, Window win) {
740     if (!frame)
            return false;

        if (frame->type == TYPE_CLIENTS) {
            int i;
745         size_t count = frame->content.clients.count;
            Window* buf = frame->content.clients.buf;
            // search for win in buf
            for (i = 0; i < count; i++) {
                if (buf[i] == win) {
750                 // if found, set focus to it
                    frame->content.clients.selection = i;
                    return true;
                }
            }
755         return false;
        } else {
            // type == TYPE_FRAMES
            // search in subframes
            bool found = frame_focus_window(frame->content.layout.a, win);
760         if (found) {
                // set selection to first frame
                frame->content.layout.selection = 0;
                return true;
            }
765         found = frame_focus_window(frame->content.layout.b, win);
            if (found) {
                // set selection to second frame
                frame->content.layout.selection = 1;
                return true;
770         }
            return false;
        }
    }

775 // focus a window
    // switch_tag if switch tag to focus to window
    // switch_monitor if switch monitor to focus to window
    // returns if window was focused or not
    bool focus_window(Window win, bool switch_tag, bool switch_monitor) {
780     HSClient* client = get_client_from_window(win);
        if (!client) return false;

        HSTag* tag = client->tag;
        assert(client->tag);
785     HSMonitor* monitor = find_monitor_with_tag(tag);
        HSMonitor* cur_mon = get_current_monitor();
        if (monitor != cur_mon && !switch_monitor) {
            // if we are not allowed to switch tag and tag is not on
            // current monitor (or on no monitor) then we cannot focus the window
790         return false;
```

```
        }
        if (monitor == NULL && !switch_tag)
            return false;

795     if (monitor != cur_mon && monitor != NULL) {
            if (!switch_monitor) {
                return false;
            } else {
                // switch monitor
800             monitor_focus_by_index(monitor_index_of(monitor));
                cur_mon = get_current_monitor();
                assert(cur_mon == monitor);
            }
        }
805     monitor_set_tag(cur_mon, tag);
        cur_mon = get_current_monitor();
        if (cur_mon->tag != tag)
            return false;

810     // now the right tag is visible, now focus it
        bool found = frame_focus_window(tag->frame, win);
        frame_focus_recursive(tag->frame);
        monitor_apply_layout(cur_mon);
        return found;
815 }

    int frame_focus_recursive(HSFrame* frame) {
        // follow the selection to a leave
        while (frame->type == TYPE_FRAMES) {
820         frame = (frame->content.layout.selection == 0) ?
                    frame->content.layout.a : frame->content.layout.b;
        }
        g_cur_frame = frame;
        if (frame->content.clients.count) {
825         int selection = frame->content.clients.selection;
            window_focus(frame->content.clients.buf[selection]);
        } else {
            window_unfocus_last();
        }
830     return 0;
    }

    // do recursive for each element of the (binary) frame tree
    // if order <= 0 -> action(node); action(left); action(right);
835 // if order == 1 -> action(left); action(node); action(right);
    // if order >= 2 -> action(left); action(right); action(node);
    void frame_do_recursive(HSFrame* frame, void (*action)(HSFrame*), int order) {
        if (!frame) return;

840     if (frame->type == TYPE_FRAMES) {
            // clients and subframes
            HSLayout* layout = &(frame->content.layout);
            if (order <= 0) action(frame);
            frame_do_recursive(layout->a, action, order);
845         if (order == 1) action(frame);
            frame_do_recursive(layout->b, action, order);
            if (order >= 2) action(frame);
        } else {
            // action only
850         action(frame);
        }
    }

    static void frame_hide(HSFrame* frame) {
855     frame_set_visible(frame, false);
        if (frame->type == TYPE_CLIENTS) {
            int i;
            Window* buf = frame->content.clients.buf;
            size_t count = frame->content.clients.count;
860         for (i = 0; i < count; i++)
                window_set_visible(buf[i], false);
        }
    }
```



```
865 void frame_show_clients(HSFrame* frame) {
        if (frame->type == TYPE_CLIENTS) {
            int i;
            Window* buf = frame->content.clients.buf;
            size_t count = frame->content.clients.count;
870         for (i = 0; i < count; i++)
                window_set_visible(buf[i], true);
        }
    }

875 void frame_remove(const Arg *arg){
        if (!g_cur_frame->parent) return;

        assert(g_cur_frame->type == TYPE_CLIENTS);
        HSFrame* parent = g_cur_frame->parent;
880     HSFrame* first = g_cur_frame;
        HSFrame* second;
        if (first == parent->content.layout.a) {
            second = parent->content.layout.b;
        } else {
885         assert(first == parent->content.layout.b);
            second = parent->content.layout.a;
        }
        size_t count;
        Window* wins;
890     // get all wins from first child
        frame_destroy(first, &wins, &count);
        // and insert them to other child.. inefficiently
        int i;
        for (i = 0; i < count; i++) {
895         frame_insert_window(second, wins[i]);
        }
        g_free(wins);
        XDestroyWindow(g_display, parent->window);
        // now do tree magic
900     // and make second child the new parent set parent
        second->parent = parent->parent;
        // copy all other elements
        *parent = *second;
        // fix childs' parent-pointer
905     if (parent->type == TYPE_FRAMES) {
            parent->content.layout.a->parent = parent;
            parent->content.layout.b->parent = parent;
        }
        g_free(second);
910     // re-layout
        frame_focus_recursive(parent);
        monitor_apply_layout(get_current_monitor());
    }

915 HSMonitor* get_current_monitor() {
        return &g_array_index(g_monitors, HSMonitor, g_cur_monitor);
    }

    void frame_set_visible(HSFrame* frame, bool visible) {
920     if (!frame)                             return;
        if (frame->window_visible == visible) return;

        window_set_visible(frame->window, visible);
        frame->window_visible = visible;
925 }

    // executes action for each client within frame and its subframes
    // if action fails (i.e. returns something != 0), then it abborts with this code
    int frame_foreach_client(HSFrame* frame, ClientAction action, void* data) {
930     int status;
        if (frame->type == TYPE_FRAMES) {
            status = frame_foreach_client(frame->content.layout.a, action, data);
            if (0 != status)   return status;

935         status = frame_foreach_client(frame->content.layout.b, action, data);
            if (0 != status)   return status;
```

```
        } else {
            // frame->type == TYPE_CLIENTS
            Window* buf = frame->content.clients.buf;
            size_t count = frame->content.clients.count;
            HSClient* client;
            for (int i = 0; i < count; i++) {
                client = get_client_from_window(buf[i]);
                // do action for each client
                status = action(client, data);
                if (0 != status)
                    return status;
            }
        }
        return 0;
    }

    void all_monitors_apply_layout() {
        int i;
        for (i = 0; i < g_monitors->len; i++) {
            HSMonitor* m = &g_array_index(g_monitors, HSMonitor, i);
            monitor_apply_layout(m);
        }
    }

    void monitor_set_tag(HSMonitor* monitor, HSTag* tag) {
        HSMonitor* other = find_monitor_with_tag(tag);
        if (monitor == other) return;
        if (other != NULL) return;

        HSTag* old_tag = monitor->tag;
        // 1. hide old tag
        frame_do_recursive(old_tag->frame, frame_hide, 2);
        // 2. show new tag
        monitor->tag = tag;
        // first reset focus and arrange windows
        frame_focus_recursive(tag->frame);
        monitor_apply_layout(monitor);
        // then show them (should reduce flicker)
        frame_do_recursive(tag->frame, frame_show_clients, 2);
        // focus window just has been shown
        // focus again to give input focus
        frame_focus_recursive(tag->frame);
    }

    void use_tag(const Arg *arg) {
       int tagindex = arg->i;

       HSMonitor* monitor = get_current_monitor();
       HSTag*  tag = find_tag(tags[tagindex]);
       if (monitor && tag)
          monitor_set_tag(get_current_monitor(), tag);
    }

    void move_tag(const Arg *arg) {
       int tagindex = arg->i;
       HSTag* target = find_tag(tags[tagindex]);
       if (!target)  return;

       tag_move_window(target);
    }

    void tag_move_window(HSTag* target) {
        HSFrame*  frame = g_cur_frame;
        if (!g_cur_frame)   return;

        Window window = frame_focused_window(frame);
        if (window == 0)    return;

        HSMonitor* monitor = get_current_monitor();
        if (monitor->tag == target)  return;

        HSMonitor* monitor_target = find_monitor_with_tag(target);
        frame_remove_window(frame, window);
```



```
        // insert window into target
        frame_insert_window(target->frame, window);
        HSClient* client = get_client_from_window(window);
        assert(client != NULL);
        client->tag = target;

        // refresh things
        if (monitor && !monitor_target) {
            // window is moved to unvisible tag so hide it
            window_set_visible(window, false);
        }
        frame_focus_recursive(frame);
        monitor_apply_layout(monitor);
        if (monitor_target)
            monitor_apply_layout(monitor_target);
    }

    void focus_monitor(const Arg *arg) {
        int new_selection = arg->i;
        // really change selection
        monitor_focus_by_index(new_selection);
    }

    int monitor_index_of(HSMonitor* monitor) {
        return monitor - (HSMonitor*)g_monitors->data;
    }

    void monitor_focus_by_index(int new_selection) {
        new_selection = CLAMP(new_selection, 0, g_monitors->len - 1);
        HSMonitor* old = &g_array_index(g_monitors, HSMonitor, g_cur_monitor);
        HSMonitor* monitor = &g_array_index(g_monitors, HSMonitor, new_selection);
        if (old == monitor)    return;

        // change selection globals
        assert(monitor->tag);
        assert(monitor->tag->frame);
        g_cur_monitor = new_selection;
        frame_focus_recursive(monitor->tag->frame);
        // repaint monitors
        monitor_apply_layout(old);
        monitor_apply_layout(monitor);
        int rx, ry;
        {
            // save old mouse position
            Window win, child;
            int wx, wy;
            unsigned int mask;
            if (True == XQueryPointer(g_display, g_root, &win, &child,
                &rx, &ry, &wx, &wy, &mask)) {
                old->mouse.x = rx - old->rect.x;
                old->mouse.y = ry - old->rect.y;
                old->mouse.x = CLAMP(old->mouse.x, 0, old->rect.width-1);
                old->mouse.y = CLAMP(old->mouse.y, 0, old->rect.height-1);
            }
        }
        // restore position of new monitor
        // but only if mouse pointer is not already on new monitor
        int new_x, new_y;
        if ((monitor->rect.x <= rx) && (rx < monitor->rect.x + monitor->rect.width)
            && (monitor->rect.y <= ry) && (ry < monitor->rect.y + monitor->rect.heig
ht)) {
            // mouse already is on new monitor
        } else {
            new_x = monitor->rect.x + monitor->mouse.x;
            new_y = monitor->rect.y + monitor->mouse.y;
            XWarpPointer(g_display, None, g_root, 0, 0, 0, 0, new_x, new_y);
        }
    }

    void create_bar(HSMonitor *mon) {
       XSetWindowAttributes wa = {
          .override_redirect = True,
          .background_pixel = dc.colors[0][ColBG],
```

```
        .background_pixmap = ParentRelative,
        .event_mask = ButtonPressMask|ExposureMask
    };

    int width = mon->rect.width;
    if(mon->primary==1) width -= systray_width;
    mon->barwin = XCreateWindow(g_display, g_root,
          mon->rect.x, mon->rect.y, width, bh, 0,
          DefaultDepth(g_display, DefaultScreen(g_display)),
          CopyFromParent,
          DefaultVisual(g_display, DefaultScreen(g_display)),
          CWOverrideRedirect|CWBackPixmap|CWEventMask, &wa);

    XMapWindow(g_display, mon->barwin);
}

void drawborder(unsigned long col[ColLast]){
    XGCValues gcv;
    XRectangle r = {dc.x, dc.y, dc.w, 2 };

    gcv.foreground = col[ColWindowBorder];
    XChangeGC(g_display, dc.gc, GCForeground, &gcv);
    XFillRectangles(g_display, dc.drawable, dc.gc, &r, 1);
}

void drawcoloredtext(char *text, HSMonitor* mon){
    Bool first=True;
    char *buf = text, *ptr = buf, c = 1;
    unsigned long *col = dc.colors[0];
    int i, ox = dc.x;

    while( *ptr ) {
       for( i = 0; *ptr < 0 || *ptr > NUMCOLORS; i++, ptr++);
       if( !*ptr ) break;
       c=*ptr;
       *ptr=0;
       if( i ) {
          dc.w = mon->rect.width - dc.x;
          drawtext(buf, col);
          dc.x += textnw(buf, i) + textnw(&c,1);
          if( first ) dc.x += (dc.font.ascent + dc.font.descent ) /2;
          first = False;
       } else if( first ) {
          ox = dc.x += textnw(&c, 1);
       }
       *ptr = c;
       col = dc.colors[ c-1 ];
       buf = ++ptr;
    }
    drawtext(buf, col);
    dc.x = ox;
}

void drawtext(const char *text, unsigned long col[ColLast]){
    char buf[256];
    int i, x, y, h, len, olen;

    XSetForeground(g_display, dc.gc, col[ColBG]);
    XFillRectangle(g_display, dc.drawable, dc.gc, dc.x, dc.y+2, dc.w, dc.h);
    if(!text)   return;

    olen = strlen(text);
    h = dc.font.ascent + dc.font.descent;
    y = dc.y + 2 + (dc.h / 2) - (h/2) + dc.font.ascent;
    x = dc.x + (h/2);

    // shorten text if necessary
    for(len = MIN(olen, sizeof buf); len && textnw(text, len) > dc.w-h; len--);
    if(!len) return;
    memcpy(buf, text, len);
    if(len < olen)
       for(i = len; i && i > len-3; buf[--i] = '.');
    XSetForeground(g_display, dc.gc, col[ColFG]);
```



```
    if(dc.font.set)
       XmbDrawString(g_display, dc.drawable, dc.font.set, dc.gc, x, y, buf, len);
    else
       XDrawString(g_display, dc.drawable, dc.gc, x, y, buf, len);
}

void initfont(const char *fontstr) {
    char *def, **missing;
    int n;

    dc.font.set = XCreateFontSet(g_display, fontstr, &missing, &n, &def);
    if(missing) {
       while(n--)
          fprintf(stderr, "fusionwm: missing fontset: %s\n", missing[n]);
       XFreeStringList(missing);
    }
    if(dc.font.set) {
       XFontStruct **xfonts;
       char **font_names;

       dc.font.ascent = dc.font.descent = 0;
       XExtentsOfFontSet(dc.font.set);
       n = XFontsOfFontSet(dc.font.set, &xfonts, &font_names);
       while(n--) {
          dc.font.ascent = MAX(dc.font.ascent, (*xfonts)->ascent);
          dc.font.descent = MAX(dc.font.descent,(*xfonts)->descent);
          xfonts++;
       }
    } else {
       if(!(dc.font.xfont = XLoadQueryFont(g_display, fontstr))
             && !(dc.font.xfont = XLoadQueryFont(g_display, "fixed")))
          die("error, cannot load font: '%s'\n", fontstr);
       dc.font.ascent = dc.font.xfont->ascent;
       dc.font.descent = dc.font.xfont->descent;
    }
    dc.font.height = dc.font.ascent + dc.font.descent;
}

int textnw(const char *text, unsigned int len) {
    XRectangle r;

    if(dc.font.set) {
       XmbTextExtents(dc.font.set, text, len, NULL, &r);
       return r.width;
    }
    return XTextWidth(dc.font.xfont, text, len);
}

int get_textw(const char *text){
    int textw = textnw(text, strlen(text)) + dc.font.height;
    return textw;
}

HSMonitor* wintomon(Window w){
    int x, y;
    HSClient *c;
    HSMonitor *m, *r;
    int di;
    unsigned int dui;
    Window dummy;
    int a, area = 0;

    if(w == g_root && XQueryPointer(g_display, g_root, &dummy, &dummy, &x, &y, &d
i, &di, &dui)){
       m = get_current_monitor();

       for(int i=0; i<g_monitors->len; i++){
          r = &g_array_index(g_monitors, HSMonitor, i);
          a = MAX(0, MIN(x+1,r->rect.x+r->rect.width) - MAX(x,r->rect.x))
              * MAX(0, MIN(y+1,r->rect.y+r->rect.height) - MAX(y,r->rect.y));
          if(a > area) {
             area = a;
             m = r;
```

```
            }
         }
         return m;
    }
    for(int i=0; i<g_monitors->len; i++){
       m = &g_array_index(g_monitors, HSMonitor, i);
       if(w == m->barwin) return m;
    }
    if((c = get_client_from_window(w))){
       m = find_monitor_with_tag(c->tag);
       return m;
    }
    return get_current_monitor();
}

void updatestatus(void) {
   if(!gettextprop(g_root, XA_WM_NAME, stext, sizeof(stext)))
      strcpy(stext, "fusionwm-"VERSION);
   draw_bar(get_current_monitor());
}

void draw_bars(){
   for(int i=0; i<g_monitors->len; i++){
      HSMonitor*m = &g_array_index(g_monitors, HSMonitor, i);
      draw_bar(m);
   }
}

void draw_bar(HSMonitor* mon){
   unsigned long *col, *bordercol;
   char separator[] = "|";
   dc.x = 0;
   dc.w = mon->rect.width;
   drawborder(dc.colors[2]);
   int barwidth = mon->rect.width;
   barwidth -= (mon->primary) ? systray_width : 0;
   HSTag* thistag;

   // Draw tag names
   for(int i=0; i < LENGTH(tags); i++){
      dc.w = get_textw(tags[i]);
      thistag = find_tag(tags[i]);
      col = dc.colors[0];
      bordercol = dc.colors[2];
      if(thistag->frame->content.clients.count > 0 ) col = dc.colors[1];
      if(thistag->urgent) col = dc.colors[3];
      if(!strcmp(tags[i], mon->tag->name->str)){
         col = dc.colors[1];
         bordercol = dc.colors[1];
      }
      drawtext(tags[i], col);
      drawborder(bordercol);
      dc.x += dc.w;
   }
   dc.w = get_textw(separator);
   drawtext(separator, dc.colors[1]);
   dc.x+= dc.w;

   // status text
   int x = dc.x;
   if(mon->primary){
      dc.w = get_textw(stext) + 5;
      dc.x = barwidth - dc.w;
      if(dc.x < x) {
         dc.x = x;
         dc.w = mon->rect.width - x;
      }
      drawcoloredtext(stext, mon);
   } else
      dc.x = mon->rect.width;

   // window title
   if((dc.w = dc.x - x) > bh) {
```



```
         dc.x = x;
         Window win = frame_focused_window(mon->tag->frame);
         char* client_title;
         if(!win || mon != get_current_monitor())
            client_title = "";
         else
            client_title = get_client_from_window(win)->title;
         drawtext(client_title, dc.colors[0]);
      }

   XCopyArea(g_display, dc.drawable, mon->barwin, dc.gc, 0, 0, barwidth, bh, 0,
0);
   XSync(g_display, False);
}
```

```
/*
 * FusionWM Main Code - main.c
 */

#include "clientlist.h"
#include "inputs.h"
#include "layout.h"
#include "globals.h"
#include "layout.h"
#include "config.h"

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <getopt.h>
#include <signal.h>
#include <sys/wait.h>
#include <assert.h>

static int (*g_xerrorxlib)(Display *, XErrorEvent *);
static unsigned int numlockmask = 0;

// handler for X-Events
void buttonpress(XEvent* event);
void buttonrelease(XEvent* event);
void configurerequest(XEvent* event);
void configurenotify(XEvent* event);
void clientmessage(XEvent* event);
void destroynotify(XEvent* event);
void enternotify(XEvent* event);
void keypress(XEvent* event);
void mappingnotify(XEvent* event);
void motionnotify(XEvent* event);
void mapnotify(XEvent *event);
void maprequest(XEvent* event);
void propertynotify(XEvent* event);
void unmapnotify(XEvent* event);
void expose(XEvent* event);

enum { ClkTagBar, ClkLtSymbol, ClkStatusText, ClkWinTitle,
       ClkClientWin, ClkRootWin, ClkLast };             /* clicks */

// handle x-events:
void event_on_configure(XEvent event) {
    XConfigureRequestEvent* cre = &event.xconfigurerequest;
    HSClient* client = get_client_from_window(cre->window);
    XConfigureEvent ce;
    ce.type = ConfigureNotify;
    ce.display = g_display;
    ce.event = cre->window;
    ce.window = cre->window;
    if (client) {
        ce.x = client->last_size.x;
        ce.y = client->last_size.y;
        ce.width = client->last_size.width;
        ce.height = client->last_size.height;
        ce.override_redirect = False;
        ce.border_width = cre->border_width;
        ce.above = cre->above;
        // FIXME: why send event and not XConfigureWindow or XMoveResizeWindow??
        XSendEvent(g_display, cre->window, False, StructureNotifyMask, (XEvent*)
&ce);
    } else {
        // if client not known.. then allow configure.
        // its probably a nice conky or dzen2 bar :)
        XWindowChanges wc;
        wc.x = cre->x;
        wc.y = cre->y;
        wc.width = cre->width;
        wc.height = cre->height;
        wc.border_width = cre->border_width;
        wc.sibling = cre->above;
```



```
        wc.stack_mode = cre->detail;
        XConfigureWindow(g_display, cre->window, cre->value_mask, &wc);
    }
}

/* There's no way to check accesses to destroyed windows, thus those cases are
 * ignored (especially on UnmapNotify's).  Other types of errors call Xlibs
 * default error handler, which may call exit.  */
int xerror(Display *dpy, XErrorEvent *ee) {
    if(ee->error_code == BadWindow
    || (ee->request_code == X_SetInputFocus && ee->error_code == BadMatch)
    || (ee->request_code == X_PolyText8 && ee->error_code == BadDrawable)
    || (ee->request_code == X_PolyFillRectangle && ee->error_code == BadDrawable
)
    || (ee->request_code == X_PolySegment && ee->error_code == BadDrawable)
    || (ee->request_code == X_ConfigureWindow && ee->error_code == BadMatch)
    || (ee->request_code == X_GrabButton && ee->error_code == BadAccess)
    || (ee->request_code == X_GrabKey && ee->error_code == BadAccess)
    || (ee->request_code == X_CopyArea && ee->error_code == BadDrawable))
        return 0;
    fprintf(stderr, "fusionwm: fatal error: request code=%d, error code=%d\n",
            ee->request_code, ee->error_code);
    if (ee->error_code == BadDrawable)
        return 0;

    return g_xerrorxlib(dpy, ee); /* may call exit */
}

int xerrordummy(Display *dpy, XErrorEvent *ee) {
   return 0;
}

/* Startup Error handler to check if another window manager is already running.
*/
int xerrorstart(Display *dpy, XErrorEvent *ee) {
   die("fusionwm: abother window manager is already running\n");
   return -1;
}

void checkotherwm(void) {
    g_xerrorxlib = XSetErrorHandler(xerrorstart);
    /* this causes an error if some other window manager is running */
    XSelectInput(g_display, DefaultRootWindow(g_display), SubstructureRedirectMa
sk);
    XSync(g_display, False);
    XSetErrorHandler(xerror);
    XSync(g_display, False);
}

// scan for windows and add them to the list of managed clients
void scan(void) {
    unsigned int i, num;
    Window d1, d2, *wins = NULL;
    XWindowAttributes wa;

    if(XQueryTree(g_display, g_root, &d1, &d2, &wins, &num)) {
        for(i = 0; i < num; i++) {
            if(!XGetWindowAttributes(g_display, wins[i], &wa)
            || wa.override_redirect || XGetTransientForHint(g_display, wins[i],
&d1))
                continue;
            if (wa.map_state == IsViewable)
                manage_client(wins[i]);
        }
        for(i = 0; i < num; i++){ // now the transients
           if(!XGetWindowAttributes(g_display, wins[i], &wa))
              continue;
           if(XGetTransientForHint(g_display, wins[i], &d1)
              && (wa.map_state == IsViewable))
              manage_client(wins[i]);
        }
        if(wins) XFree(wins);
    }
```

```
    }

    void sigchld(int unused){
       if(signal(SIGCHLD, sigchld) == SIG_ERR)
          die("Can't install SIGCHLD handler");
       while(0 < waitpid(-1, NULL, WNOHANG));
    }

    static void (*handler[LASTEvent]) (XEvent *) = {
       [ ButtonPress       ] = buttonpress,
       [ ButtonRelease     ] = buttonrelease,
       [ ClientMessage     ] = ewmh_handle_client_message,
       [ ConfigureRequest  ] = configurerequest,
       [ ConfigureNotify   ] = configurenotify,
       [ DestroyNotify     ] = destroynotify,
       [ EnterNotify       ] = enternotify,
       [ Expose            ] = expose,
       [ KeyPress          ] = keypress,
       [ MappingNotify     ] = mappingnotify,
       [ MotionNotify      ] = motionnotify,
       [ MapNotify         ] = mapnotify,
       [ MapRequest        ] = maprequest,
       [ PropertyNotify    ] = propertynotify,
       [ UnmapNotify       ] = unmapnotify
    };

    // event handler implementations
    void buttonpress(XEvent* event) {
       XButtonEvent* be = &(event->xbutton);
       unsigned int i, x, click;
       Arg arg = {0};
       HSMonitor *m;

       // focus monitor if necessary
       if((m = wintomon(be->window)) && m != get_current_monitor()) {
          monitor_focus_by_index(monitor_index_of(m));
       }
       if (be->window == g_root && be->subwindow != None) {
          if (mouse_binding_find(be->state, be->button)) {
             mouse_start_drag(event);
          }
       } else {
          if(be->window == get_current_monitor()->barwin){
             i = x = 0;
             do
                x += get_textw(tags[i]);
             while(be->x >= x && ++i < LENGTH(tags));
             if(i < LENGTH(tags)){
                click = ClkTagBar;
                arg.i = i;
             } else
                click = ClkWinTitle;
          }
          if(click == ClkTagBar && be->button == Button1 && CLEANMASK(be->state) ==
    0)
             use_tag(&arg);

          if (be->button == Button1 || be->button == Button2 || be->button == Button
    3) {
             // only change focus on real clicks... not when scrolling
             if (raise_on_click)
                XRaiseWindow(g_display, be->window);

             focus_window(be->window, false, true);
          }
          // handling of event is finished, now propagate event to window
          XAllowEvents(g_display, ReplayPointer, CurrentTime);
       }
    }

    void buttonrelease(XEvent* event) {
        mouse_stop_drag();
    }
```



```
    void configurerequest(XEvent* event) {
        event_on_configure(*event);
    }

    void configurenotify(XEvent* event){
       XConfigureEvent *ev = &event->xconfigure;
       HSMonitor *m = get_current_monitor();

       if(ev->window == g_root)
          XMoveResizeWindow(g_display, m->barwin, m->rect.x, m->rect.y, m->rect.widt
    h, bh);
    }

    void clientmessage(XEvent* event) {
        ewmh_handle_client_message(event);
    }

    void destroynotify(XEvent* event) {
        // try to unmanage it
        unmanage_client(event->xdestroywindow.window);
    }

    void enternotify(XEvent* event) {
        if (focus_follows_mouse && !event->xcrossing.focus)
            focus_window(event->xcrossing.window, false, true); // sloppy focus
    }

    void expose(XEvent* event){
       XExposeEvent *ev = &event->xexpose;
       if(ev->count == 0) draw_bar(get_current_monitor());
    }

    void keypress(XEvent* event) {
        key_press(event);
    }

    void mappingnotify(XEvent* event) {
       // regrab when keyboard map changes
       XMappingEvent *ev = &event->xmapping;
       XRefreshKeyboardMapping(ev);
       if(ev->request == MappingKeyboard) {
          grab_keys();
          grab_buttons();
       }
    }

    void motionnotify(XEvent* event) {
        handle_motion_event(event);
    }

    void mapnotify(XEvent* event) {
       HSClient* c;
       if ((c = get_client_from_window(event->xmap.window))) {
          // reset focus. so a new window gets the focus if it shall have the input
    focus
          frame_focus_recursive(g_cur_frame);
          // also update the window title - just to be sure
          client_update_title(c);
       }
    }

    void maprequest(XEvent* event) {
        XMapRequestEvent* mapreq = &event->xmaprequest;
        if (!get_client_from_window(mapreq->window)) {
            // client should be managed (is not ignored)
            // but is not managed yet
            HSClient* client = manage_client(mapreq->window);
            if (client && find_monitor_with_tag(client->tag))
                XMapWindow(g_display, mapreq->window);
        }
    }
```

```
void propertynotify(XEvent* event) {
    XPropertyEvent *ev = &event->xproperty;
    HSClient* client;
    if((ev->window == g_root) && (ev->atom == XA_WM_NAME))
         updatestatus();

    if (ev->state == PropertyNewValue &&
          (client = get_client_from_window(ev->window))) {
       switch (ev->atom) {
          case XA_WM_HINTS:
             client_update_wm_hints(client);
             break;
          case XA_WM_NAME:
             client_update_title(client);
             break;
          default:
             break;
        }
    }
    draw_bars();
}

void unmapnotify(XEvent* event) {
    unmanage_client(event->xunmap.window);
}

void setup(void){
   // remove zombies on SIGCHLD
   sigchld(0);

   // set some globals
   g_screen = DefaultScreen(g_display);
   g_screen_width = DisplayWidth(g_display, g_screen);
   g_screen_height = DisplayHeight(g_display, g_screen);
   g_root = RootWindow(g_display, g_screen);
   XSelectInput(g_display, g_root, ROOT_EVENT_MASK);

   // initialize subsystems
   inputs_init();
   clientlist_init();
   layout_init();
}

void cleanup(void) {
   inputs_destroy();
   clientlist_destroy();
   layout_destroy();
}

// --- Main Function ------------------------------------------------------
---------
int main(int argc, char* argv[]) {

   if(argc == 2 && !strcmp("-v", argv[1]))
      die("fusionwm-"VERSION" \n");
   else if (argc != 1)
      die("usage: fusionwm [-v]\n");

   if(!setlocale(LC_CTYPE, "") || !XSupportsLocale())
      fputs("warning: no locale support\n", stderr);

   if(!(g_display = XOpenDisplay(NULL)))
      die("fusionwm: cannot open display\n");
   checkotherwm();

   // Setup
   setup();
   scan();
   all_monitors_apply_layout();
   updatestatus();

   // Main loop
   XEvent event;
```



```
   XSync(g_display, False);
   while (!g_aboutToQuit && !XNextEvent(g_display, &event))
      if(handler[event.type])
         handler[event.type](&event); // call handler

   // Cleanup
   cleanup();
   XCloseDisplay(g_display);

   return EXIT_SUCCESS;
}
```