```
/* See LICENSE file for copyright and license details. */

/* appearance */
static const char font[]            = "-*-terminus-medium-r-*-*-17-*-*-*-*-*-*-*";
static const char normbordercolor[] = "#009696";
static const char nobordercolor[] = "#000000";
static const char normbgcolor[]     = "#091720";
static const char normfgcolor[]     = "#74b4a6";
static const char selbordercolor[]  = "#7affff";
static const char selbgcolor[]      = "#091720";
static const char selfgcolor[]      = "#addaed";
static const unsigned int borderpx  = 1;        /* border pixel of windows */
static const unsigned int snap      = 32;       /* snap pixel */
static const Bool showbar           = True;     /* False means no bar */
static const Bool topbar            = True;     /* False means bottom bar */

/* tagging */
static const char *tags[] = { "1", "2", "3", "4", "5", "6", "7", "8", "9" };

static const Rule rules[] = {
   /* xprop(1):
    * WM_CLASS(STRING) = instance, class
    * WM_NAME(STRING) = title
    */
   /* class      instance    title       tags mask     isfloating   monitor */
   { "Gimp",     NULL,       NULL,       0,            True,        -1 },
   { "Firefox",  NULL,       NULL,       1 << 8,       False,       -1 },
};

/* layout(s) */
static const float mfact      = 0.55; /* factor of master area size [0.05..0.95] */
static const int nmaster      = 1;    /* number of clients in master area */
static const Bool resizehints = True; /* True means respect size hints in tiled resizals */

static const Layout layouts[] = {
   /* symbol     arrange function */
   { "[]=",      tile },    /* first entry is default */
   { "><>",      NULL },    /* no layout function means floating behavior */
   { "[M]",      monocle },
};

/* key definitions */
#define MODKEY Mod1Mask
#define TAGKEYS(KEY,TAG) \
   { MODKEY,                       KEY,      view,           {.ui = 1 << TAG} }, \
   { MODKEY|ControlMask,           KEY,      toggleview,     {.ui = 1 << TAG} }, \
   { MODKEY|ShiftMask,             KEY,      tag,            {.ui = 1 << TAG} }, \
   { MODKEY|ControlMask|ShiftMask, KEY,      toggletag,      {.ui = 1 << TAG} },

/* helper for spawning shell commands in the pre dwm-5.0 fashion */
#define SHCMD(cmd) { .v = (const char*[]){ "/bin/sh", "-c", cmd, NULL } }

/* commands */
static char dmenumon[2] = "0"; /* component of dmenucmd, manipulated in spawn() */
static const char *dmenucmd[] = { "dmenu_run", "-m", dmenumon, "-fn", font, "-nb",
   normbgcolor, "-nf", normfgcolor, "-sb", selbgcolor, "-sf", selfgcolor, NULL };
static const char *xboomxcmd[] = { "xboomx", NULL };
static const char *termcmd[]  = { "uxterm", NULL };

static Key keys[] = {
   /* modifier                     key        function        argument */
   { MODKEY,                       XK_p,      spawn,          {.v = xboomxcmd } },
   //{ MODKEY,                       XK_p,      spawn,          {.v = dmenucmd } },
   { MODKEY|ShiftMask,             XK_Return, spawn,          {.v = termcmd } },
   { MODKEY,                       XK_b,      togglebar,      {0} },
   { MODKEY,                       XK_j,      focusstack,     {.i = +1 } },
   { MODKEY,                       XK_k,      focusstack,     {.i = -1 } },
   { MODKEY,                       XK_i,      incnmaster,     {.i = +1 } },
   { MODKEY,                       XK_d,      incnmaster,     {.i = -1 } },
   { MODKEY,                       XK_h,      setmfact,       {.f = -0.05} },
   { MODKEY,                       XK_l,      setmfact,       {.f = +0.05} },
```

```
    { MODKEY,                       XK_Return, zoom,           {0} },
    { MODKEY,                       XK_Tab,    view,           {0} },
    { MODKEY|ShiftMask,             XK_c,      killclient,     {0} },
    { MODKEY,                       XK_t,      setlayout,      {.v = &layouts[0]} },
    { MODKEY,                       XK_f,      setlayout,      {.v = &layouts[1]} },
    { MODKEY,                       XK_m,      setlayout,      {.v = &layouts[2]} },
    { MODKEY,                       XK_space,  setlayout,      {0} },
    { MODKEY|ShiftMask,             XK_space,  togglefloating, {0} },
    { MODKEY,                       XK_0,      view,           {.ui = ~0 } },
    { MODKEY|ShiftMask,             XK_0,      tag,            {.ui = ~0 } },
    { MODKEY,                       XK_comma,  focusmon,       {.i = -1 } },
    { MODKEY,                       XK_period, focusmon,       {.i = +1 } },
    { MODKEY|ShiftMask,             XK_comma,  tagmon,         {.i = -1 } },
    { MODKEY|ShiftMask,             XK_period, tagmon,         {.i = +1 } },
    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)
    TAGKEYS(                        XK_8,                      7)
    TAGKEYS(                        XK_9,                      8)
    { MODKEY|ShiftMask,             XK_q,      quit,           {0} },
};

/* button definitions */
/* click can be ClkLtSymbol, ClkStatusText, ClkWinTitle, ClkClientWin, or ClkRootWin */
static Button buttons[] = {
    /* click                event mask      button          function        argument */
    { ClkLtSymbol,          0,              Button1,        setlayout,      {0} },
    { ClkLtSymbol,          0,              Button3,        setlayout,      {.v = &layouts[2]} },
    { ClkWinTitle,          0,              Button2,        zoom,           {0} },
    { ClkStatusText,        0,              Button2,        spawn,          {.v = termcmd } },
    { ClkClientWin,         MODKEY,         Button1,        movemouse,      {0} },
    { ClkClientWin,         MODKEY,         Button2,        togglefloating, {0} },
    { ClkClientWin,         MODKEY,         Button3,        resizemouse,    {0} },
    { ClkTagBar,            0,              Button1,        view,           {0} },
    { ClkTagBar,            0,              Button3,        toggleview,     {0} },
    { ClkTagBar,            MODKEY,         Button1,        tag,            {0} },
    { ClkTagBar,            MODKEY,         Button3,        toggletag,      {0} },
};
```

```
/* See LICENSE file for copyright and license details.
 *
 * dynamic window manager is designed like any other X client as well. It is
 * driven through handling X events. In contrast to other X clients, a window
 * manager selects for SubstructureRedirectMask on the root window, to receive
 * events about window (dis-)appearance.  Only one X connection at a time is
 * allowed to select for this event mask.
 *
 * The event handlers of dwm are organized in an array which is accessed
 * whenever a new event has been fetched. This allows event dispatching
 * in O(1) time.
 *
 * Each child of the root window is called a client, except windows which have
 * set the override_redirect flag.  Clients are organized in a linked client
 * list on each monitor, the focus history is remembered through a stack list
 * on each monitor. Each client contains a bit array to indicate the tags of a
 * client.
 *
 * Keys and tagging rules are organized as arrays and defined in config.h.
 *
 * To understand everything else, start reading main().
 */
#include <errno.h>
#include <locale.h>
#include <stdarg.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <X11/cursorfont.h>
#include <X11/keysym.h>
#include <X11/Xatom.h>
#include <X11/Xlib.h>
#include <X11/Xproto.h>
#include <X11/Xutil.h>
#ifdef XINERAMA
#include <X11/extensions/Xinerama.h>
#endif /* XINERAMA */

#include "drw.h"
#include "util.h"

/* macros */
#define BUTTONMASK              (ButtonPressMask|ButtonReleaseMask)
#define CLEANMASK(mask)         (mask & ~(numlockmask|LockMask) &
   (ShiftMask|ControlMask|Mod1Mask|Mod2Mask|Mod3Mask|Mod4Mask|Mod5Mask))
#define INTERSECT(x,y,w,h,m)    (MAX(0, MIN((x)+(w),(m)->wx+(m)->ww) - MAX((x),(m)->wx)) \
                                * MAX(0, MIN((y)+(h),(m)->wy+(m)->wh) - MAX((y),(m)->wy)))
#define ISVISIBLE(C)            ((C->tags & C->mon->tagset[C->mon->seltags]))
#define LENGTH(X)               (sizeof X / sizeof X[0])
#define MOUSEMASK               (BUTTONMASK|PointerMotionMask)
#define WIDTH(X)                ((X)->w + 2 * (X)->bw)
#define HEIGHT(X)               ((X)->h + 2 * (X)->bw)
#define TAGMASK                 ((1 << LENGTH(tags)) - 1)
#define TEXTW(X)                (drw_font_getexts_width(drw->font, X, strlen(X)) + drw->font->h)

/* enums */
enum { CurNormal, CurResize, CurMove, CurLast }; /* cursor */
enum { SchemeNorm, SchemeSel, SchemeLast }; /* color schemes */
enum { NetSupported, NetWMName, NetWMState,
       NetWMFullscreen, NetActiveWindow, NetWMWindowType,
       NetWMWindowTypeDialog, NetClientList, NetLast }; /* EWMH atoms */
enum { WMProtocols, WMDelete, WMState, WMTakeFocus, WMLast }; /* default atoms */
enum { ClkTagBar, ClkLtSymbol, ClkStatusText, ClkWinTitle,
       ClkClientWin, ClkRootWin, ClkLast }; /* clicks */
```

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

typedef struct {
    unsigned int click;
    unsigned int mask;
    unsigned int button;
    void (*func)(const Arg *arg);
    const Arg arg;
} Button;

typedef struct Monitor Monitor;
typedef struct Client Client;
struct Client {
    char name[256];
    float mina, maxa;
    int x, y, w, h;
    int oldx, oldy, oldw, oldh;
    int basew, baseh, incw, inch, maxw, maxh, minw, minh;
    int bw, oldbw;
    unsigned int tags;
    Bool isfixed, isfloating, isurgent, neverfocus, oldstate, isfullscreen;
    Client *next;
    Client *snext;
    Monitor *mon;
    Window win;
};

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

typedef struct {
    const char *symbol;
    void (*arrange)(Monitor *);
} Layout;

struct Monitor {
    char ltsymbol[16];
    float mfact;
    int nmaster;
    int num;
    int by;               /* bar geometry */
    int mx, my, mw, mh;   /* screen size */
    int wx, wy, ww, wh;   /* window area  */
    unsigned int seltags;
    unsigned int sellt;
    unsigned int tagset[2];
    Bool showbar;
    Bool topbar;
    Client *clients;
    Client *sel;
    Client *stack;
    Monitor *next;
    Window barwin;
    const Layout *lt[2];
};

typedef struct {
    const char *class;
    const char *instance;
    const char *title;
    unsigned int tags;
    Bool isfloating;
```

```
    int monitor;
} Rule;

/* function declarations */
static void applyrules(Client *c);
static Bool applysizehints(Client *c, int *x, int *y, int *w, int *h, Bool interact);
static void arrange(Monitor *m);
static void arrangemon(Monitor *m);
static void attach(Client *c);
static void attachstack(Client *c);
static void buttonpress(XEvent *e);
static void checkotherwm(void);
static void cleanup(void);
static void cleanupmon(Monitor *mon);
static void clearurgent(Client *c);
static void clientmessage(XEvent *e);
static void configure(Client *c);
static void configurenotify(XEvent *e);
static void configurerequest(XEvent *e);
static Monitor *createmon(void);
static void destroynotify(XEvent *e);
static void detach(Client *c);
static void detachstack(Client *c);
static Monitor *dirtomon(int dir);
static void drawbar(Monitor *m);
static void drawbars(void);
static void enternotify(XEvent *e);
static void expose(XEvent *e);
static void focus(Client *c);
static void focusin(XEvent *e);
static void focusmon(const Arg *arg);
static void focusstack(const Arg *arg);
static Bool getrootptr(int *x, int *y);
static long getstate(Window w);
static Bool gettextprop(Window w, Atom atom, char *text, unsigned int size);
static void grabbuttons(Client *c, Bool focused);
static void grabkeys(void);
static void incnmaster(const Arg *arg);
static void keypress(XEvent *e);
static void killclient(const Arg *arg);
static void manage(Window w, XWindowAttributes *wa);
static void mappingnotify(XEvent *e);
static void maprequest(XEvent *e);
static void monocle(Monitor *m);
static void motionnotify(XEvent *e);
static void movemouse(const Arg *arg);
static Client *nexttiled(Client *c);
static void pop(Client *);
static void propertynotify(XEvent *e);
static void quit(const Arg *arg);
static Monitor *recttomon(int x, int y, int w, int h);
static void resize(Client *c, int x, int y, int w, int h, Bool interact);
static void resizeclient(Client *c, int x, int y, int w, int h);
static void resizemouse(const Arg *arg);
static void restack(Monitor *m);
static void run(void);
static void scan(void);
static Bool sendevent(Client *c, Atom proto);
static void sendmon(Client *c, Monitor *m);
static void setclientstate(Client *c, long state);
static void setfocus(Client *c);
static void setfullscreen(Client *c, Bool fullscreen);
static void setlayout(const Arg *arg);
static void setmfact(const Arg *arg);
static void setup(void);
static void showhide(Client *c);
static void sigchld(int unused);
static void spawn(const Arg *arg);
static void tag(const Arg *arg);
static void tagmon(const Arg *arg);
static void tile(Monitor *);
static void togglebar(const Arg *arg);
```

```
static void togglefloating(const Arg *arg);
static void toggletag(const Arg *arg);
static void toggleview(const Arg *arg);
static void unfocus(Client *c, Bool setfocus);
static void unmanage(Client *c, Bool destroyed);
static void unmapnotify(XEvent *e);
static Bool updategeom(void);
static void updatebarpos(Monitor *m);
static void updatebars(void);
static void updateclientlist(void);
static void updatenumlockmask(void);
static void updatesizehints(Client *c);
static void updatestatus(void);
static void updatewindowtype(Client *c);
static void updatetitle(Client *c);
static void updatewmhints(Client *c);
static void view(const Arg *arg);
static Client *wintoclient(Window w);
static Monitor *wintomon(Window w);
static int xerror(Display *dpy, XErrorEvent *ee);
static int xerrordummy(Display *dpy, XErrorEvent *ee);
static int xerrorstart(Display *dpy, XErrorEvent *ee);
static void zoom(const Arg *arg);

/* variables */
static const char broken[] = "broken";
static char stext[256];
static int screen;
static int sw, sh;           /* X display screen geometry width, height */
static int bh, blw = 0;      /* bar geometry */
static int (*xerrorxlib)(Display *, XErrorEvent *);
static unsigned int numlockmask = 0;
static void (*handler[LASTEvent]) (XEvent *) = {
   [ButtonPress] = buttonpress,
   [ClientMessage] = clientmessage,
   [ConfigureRequest] = configurerequest,
   [ConfigureNotify] = configurenotify,
   [DestroyNotify] = destroynotify,
   [EnterNotify] = enternotify,
   [Expose] = expose,
   [FocusIn] = focusin,
   [KeyPress] = keypress,
   [MappingNotify] = mappingnotify,
   [MapRequest] = maprequest,
   [MotionNotify] = motionnotify,
   [PropertyNotify] = propertynotify,
   [UnmapNotify] = unmapnotify
};
static Atom wmatom[WMLast], netatom[NetLast];
static Bool running = True;
static Cur *cursor[CurLast];
static ClrScheme scheme[SchemeLast];
static Display *dpy;
static Drw *drw;
static Fnt *fnt;
static Monitor *mons, *selmon;
static Window root;

/* configuration, allows nested code to access above variables */
#include "config.h"

/* compile-time check if all tags fit into an unsigned int bit array. */
struct NumTags { char limitexceeded[LENGTH(tags) > 31 ? -1 : 1]; };

/* function implementations */
void
applyrules(Client *c) {
   const char *class, *instance;
   unsigned int i;
   const Rule *r;
   Monitor *m;
   XClassHint ch = { NULL, NULL };
```

```
    /* rule matching */
    c->isfloating = c->tags = 0;
    XGetClassHint(dpy, c->win, &ch);
    class    = ch.res_class ? ch.res_class : broken;
    instance = ch.res_name  ? ch.res_name  : broken;

    for(i = 0; i < LENGTH(rules); i++) {
        r = &rules[i];
        if((!r->title || strstr(c->name, r->title))
        && (!r->class || strstr(class, r->class))
        && (!r->instance || strstr(instance, r->instance)))
        {
            c->isfloating = r->isfloating;
            c->tags |= r->tags;
            for(m = mons; m && m->num != r->monitor; m = m->next);
            if(m)
                c->mon = m;
        }
    }
    if(ch.res_class)
        XFree(ch.res_class);
    if(ch.res_name)
        XFree(ch.res_name);
    c->tags = c->tags & TAGMASK ? c->tags & TAGMASK : c->mon->tagset[c->mon->seltags];
}

Bool
applysizehints(Client *c, int *x, int *y, int *w, int *h, Bool interact) {
    Bool baseismin;
    Monitor *m = c->mon;

    /* set minimum possible */
    *w = MAX(1, *w);
    *h = MAX(1, *h);
    if(interact) {
        if(*x > sw)
            *x = sw - WIDTH(c);
        if(*y > sh)
            *y = sh - HEIGHT(c);
        if(*x + *w + 2 * c->bw < 0)
            *x = 0;
        if(*y + *h + 2 * c->bw < 0)
            *y = 0;
    }
    else {
        if(*x >= m->wx + m->ww)
            *x = m->wx + m->ww - WIDTH(c);
        if(*y >= m->wy + m->wh)
            *y = m->wy + m->wh - HEIGHT(c);
        if(*x + *w + 2 * c->bw <= m->wx)
            *x = m->wx;
        if(*y + *h + 2 * c->bw <= m->wy)
            *y = m->wy;
    }
    if(*h < bh)
        *h = bh;
    if(*w < bh)
        *w = bh;
    if(resizehints || c->isfloating || !c->mon->lt[c->mon->sellt]->arrange) {
        /* see last two sentences in ICCCM 4.1.2.3 */
        baseismin = c->basew == c->minw && c->baseh == c->minh;
        if(!baseismin) { /* temporarily remove base dimensions */
            *w -= c->basew;
            *h -= c->baseh;
        }
        /* adjust for aspect limits */
        if(c->mina > 0 && c->maxa > 0) {
            if(c->maxa < (float)*w / *h)
                *w = *h * c->maxa + 0.5;
            else if(c->mina < (float)*h / *w)
                *h = *w * c->mina + 0.5;
```

```
        }
        if(baseismin) { /* increment calculation requires this */
            *w -= c->basew;
            *h -= c->baseh;
        }
        /* adjust for increment value */
        if(c->incw)
            *w -= *w % c->incw;
        if(c->inch)
            *h -= *h % c->inch;
        /* restore base dimensions */
        *w = MAX(*w + c->basew, c->minw);
        *h = MAX(*h + c->baseh, c->minh);
        if(c->maxw)
            *w = MIN(*w, c->maxw);
        if(c->maxh)
            *h = MIN(*h, c->maxh);
    }
    return *x != c->x || *y != c->y || *w != c->w || *h != c->h;
}

void
arrange(Monitor *m) {
    if(m)
        showhide(m->stack);
    else for(m = mons; m; m = m->next)
        showhide(m->stack);
    if(m) {
        arrangemon(m);
        restack(m);
    } else for(m = mons; m; m = m->next)
        arrangemon(m);
}

void
arrangemon(Monitor *m) {
    strncpy(m->ltsymbol, m->lt[m->sellt]->symbol, sizeof m->ltsymbol);
    if(m->lt[m->sellt]->arrange)
        m->lt[m->sellt]->arrange(m);
}

void
attach(Client *c) {
    c->next = c->mon->clients;
    c->mon->clients = c;
}

void
attachstack(Client *c) {
    c->snext = c->mon->stack;
    c->mon->stack = c;
}

void
buttonpress(XEvent *e) {
    unsigned int i, x, click;
    Arg arg = {0};
    Client *c;
    Monitor *m;
    XButtonPressedEvent *ev = &e->xbutton;

    click = ClkRootWin;
    /* focus monitor if necessary */
    if((m = wintomon(ev->window)) && m != selmon) {
        unfocus(selmon->sel, True);
        selmon = m;
        focus(NULL);
    }
    if(ev->window == selmon->barwin) {
        i = x = 0;
        do
            x += TEXTW(tags[i]);
```

```
        while(ev->x >= x && ++i < LENGTH(tags));
        if(i < LENGTH(tags)) {
            click = ClkTagBar;
            arg.ui = 1 << i;
        }
        else if(ev->x < x + blw)
            click = ClkLtSymbol;
        else if(ev->x > selmon->ww - TEXTW(stext))
            click = ClkStatusText;
        else
            click = ClkWinTitle;
    }
    else if((c = wintoclient(ev->window))) {
        focus(c);
        click = ClkClientWin;
    }
    for(i = 0; i < LENGTH(buttons); i++)
        if(click == buttons[i].click && buttons[i].func && buttons[i].button == ev->button
        && CLEANMASK(buttons[i].mask) == CLEANMASK(ev->state))
            buttons[i].func(click == ClkTagBar && buttons[i].arg.i == 0 ? &arg : &buttons[i].arg);
}

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

void
cleanup(void) {
    Arg a = {.ui = ~0};
    Layout foo = { "", NULL };
    Monitor *m;

    view(&a);
    selmon->lt[selmon->sellt] = &foo;
    for(m = mons; m; m = m->next)
        while(m->stack)
            unmanage(m->stack, False);
    XUngrabKey(dpy, AnyKey, AnyModifier, root);
    while(mons)
        cleanupmon(mons);
    drw_cur_free(drw, cursor[CurNormal]);
    drw_cur_free(drw, cursor[CurResize]);
    drw_cur_free(drw, cursor[CurMove]);
    drw_font_free(dpy, fnt);
    drw_clr_free(scheme[SchemeNorm].border);
    drw_clr_free(scheme[SchemeNorm].bg);
    drw_clr_free(scheme[SchemeNorm].fg);
    drw_clr_free(scheme[SchemeSel].border);
    drw_clr_free(scheme[SchemeSel].bg);
    drw_clr_free(scheme[SchemeSel].fg);
    drw_free(drw);
    XSync(dpy, False);
    XSetInputFocus(dpy, PointerRoot, RevertToPointerRoot, CurrentTime);
    XDeleteProperty(dpy, root, netatom[NetActiveWindow]);
}

void
cleanupmon(Monitor *mon) {
    Monitor *m;

    if(mon == mons)
        mons = mons->next;
    else {
        for(m = mons; m && m->next != mon; m = m->next);
        m->next = mon->next;
    }
```

```
    XUnmapWindow(dpy, mon->barwin);
    XDestroyWindow(dpy, mon->barwin);
    free(mon);
}

void
clearurgent(Client *c) {
    XWMHints *wmh;

    c->isurgent = False;
    if(!(wmh = XGetWMHints(dpy, c->win)))
        return;
    wmh->flags &= ~XUrgencyHint;
    XSetWMHints(dpy, c->win, wmh);
    XFree(wmh);
}

void
clientmessage(XEvent *e) {
    XClientMessageEvent *cme = &e->xclient;
    Client *c = wintoclient(cme->window);

    if(!c)
        return;
    if(cme->message_type == netatom[NetWMState]) {
        if(cme->data.l[1] == netatom[NetWMFullscreen] || cme->data.l[2] == netatom[NetWMFullscreen])
            setfullscreen(c, (cme->data.l[0] == 1 /* _NET_WM_STATE_ADD    */
                              || (cme->data.l[0] == 2 /* _NET_WM_STATE_TOGGLE */ && !c->isfullscreen)));
    }
    else if(cme->message_type == netatom[NetActiveWindow]) {
        if(!ISVISIBLE(c)) {
            c->mon->seltags ^= 1;
            c->mon->tagset[c->mon->seltags] = c->tags;
        }
        pop(c);
    }
}

void
configure(Client *c) {
    XConfigureEvent ce;

    ce.type = ConfigureNotify;
    ce.display = dpy;
    ce.event = c->win;
    ce.window = c->win;
    ce.x = c->x;
    ce.y = c->y;
    ce.width = c->w;
    ce.height = c->h;
    ce.border_width = c->bw;
    ce.above = None;
    ce.override_redirect = False;
    XSendEvent(dpy, c->win, False, StructureNotifyMask, (XEvent *)&ce);
}

void
configurenotify(XEvent *e) {
    Monitor *m;
    XConfigureEvent *ev = &e->xconfigure;
    Bool dirty;

    // TODO: updategeom handling sucks, needs to be simplified
    if(ev->window == root) {
        dirty = (sw != ev->width || sh != ev->height);
        sw = ev->width;
        sh = ev->height;
        if(updategeom() || dirty) {
            drw_resize(drw, sw, bh);
            updatebars();
            for(m = mons; m; m = m->next)
                XMoveResizeWindow(dpy, m->barwin, m->wx, m->by, m->ww, bh);
```

```
            focus(NULL);
            arrange(NULL);
        }
    }
}

void
configurerequest(XEvent *e) {
    Client *c;
    Monitor *m;
    XConfigureRequestEvent *ev = &e->xconfigurerequest;
    XWindowChanges wc;

    if((c = wintoclient(ev->window))) {
        if(ev->value_mask & CWBorderWidth)
            c->bw = ev->border_width;
        else if(c->isfloating || !selmon->lt[selmon->sellt]->arrange) {
            m = c->mon;
            if(ev->value_mask & CWX) {
                c->oldx = c->x;
                c->x = m->mx + ev->x;
            }
            if(ev->value_mask & CWY) {
                c->oldy = c->y;
                c->y = m->my + ev->y;
            }
            if(ev->value_mask & CWWidth) {
                c->oldw = c->w;
                c->w = ev->width;
            }
            if(ev->value_mask & CWHeight) {
                c->oldh = c->h;
                c->h = ev->height;
            }
            if((c->x + c->w) > m->mx + m->mw && c->isfloating)
                c->x = m->mx + (m->mw / 2 - WIDTH(c) / 2); /* center in x direction */
            if((c->y + c->h) > m->my + m->mh && c->isfloating)
                c->y = m->my + (m->mh / 2 - HEIGHT(c) / 2); /* center in y direction */
            if((ev->value_mask & (CWX|CWY)) && !(ev->value_mask & (CWWidth|CWHeight)))
                configure(c);
            if(ISVISIBLE(c))
                XMoveResizeWindow(dpy, c->win, c->x, c->y, c->w, c->h);
        }
        else
            configure(c);
    }
    else {
        wc.x = ev->x;
        wc.y = ev->y;
        wc.width = ev->width;
        wc.height = ev->height;
        wc.border_width = ev->border_width;
        wc.sibling = ev->above;
        wc.stack_mode = ev->detail;
        XConfigureWindow(dpy, ev->window, ev->value_mask, &wc);
    }
    XSync(dpy, False);
}

Monitor *
createmon(void) {
    Monitor *m;

    if(!(m = (Monitor *)calloc(1, sizeof(Monitor))))
        die("fatal: could not malloc() %u bytes\n", sizeof(Monitor));
    m->tagset[0] = m->tagset[1] = 1;
    m->mfact = mfact;
    m->nmaster = nmaster;
    m->showbar = showbar;
    m->topbar = topbar;
    m->lt[0] = &layouts[0];
    m->lt[1] = &layouts[1 % LENGTH(layouts)];
```

```
    strncpy(m->ltsymbol, layouts[0].symbol, sizeof m->ltsymbol);
    return m;
}

void
destroynotify(XEvent *e) {
    Client *c;
    XDestroyWindowEvent *ev = &e->xdestroywindow;

    if((c = wintoclient(ev->window)))
        unmanage(c, True);
}

void
detach(Client *c) {
    Client **tc;

    for(tc = &c->mon->clients; *tc && *tc != c; tc = &(*tc)->next);
    *tc = c->next;
}

void
detachstack(Client *c) {
    Client **tc, *t;

    for(tc = &c->mon->stack; *tc && *tc != c; tc = &(*tc)->snext);
    *tc = c->snext;

    if(c == c->mon->sel) {
        for(t = c->mon->stack; t && !ISVISIBLE(t); t = t->snext);
        c->mon->sel = t;
    }
}

Monitor *
dirtomon(int dir) {
    Monitor *m = NULL;

    if(dir > 0) {
        if(!(m = selmon->next))
            m = mons;
    }
    else if(selmon == mons)
        for(m = mons; m->next; m = m->next);
    else
        for(m = mons; m->next != selmon; m = m->next);
    return m;
}

void
drawbar(Monitor *m) {
    int x, xx, w;
    unsigned int i, occ = 0, urg = 0;
    Client *c;

    for(c = m->clients; c; c = c->next) {
        occ |= c->tags;
        if(c->isurgent)
            urg |= c->tags;
    }
    x = 0;
    for(i = 0; i < LENGTH(tags); i++) {
        w = TEXTW(tags[i]);
        drw_setscheme(drw, m->tagset[m->seltags] & 1 << i ? &scheme[SchemeSel] : &scheme[SchemeNorm]);
        drw_text(drw, x, 0, w, bh, tags[i], urg & 1 << i);
        drw_rect(drw, x, 0, w, bh, m == selmon && selmon->sel && selmon->sel->tags & 1 << i,
                 occ & 1 << i, urg & 1 << i);
        x += w;
    }
    w = blw = TEXTW(m->ltsymbol);
    drw_setscheme(drw, &scheme[SchemeNorm]);
    drw_text(drw, x, 0, w, bh, m->ltsymbol, 0);
```

```
    x += w;
    xx = x;
    if(m == selmon) { /* status is only drawn on selected monitor */
        w = TEXTW(stext);
        x = m->ww - w;
        if(x < xx) {
            x = xx;
            w = m->ww - xx;
        }
        drw_text(drw, x, 0, w, bh, stext, 0);
    }
    else
        x = m->ww;
    if((w = x - xx) > bh) {
        x = xx;
        if(m->sel) {
            drw_setscheme(drw, m == selmon ? &scheme[SchemeSel] : &scheme[SchemeNorm]);
            drw_text(drw, x, 0, w, bh, m->sel->name, 0);
            drw_rect(drw, x, 0, w, bh, m->sel->isfixed, m->sel->isfloating, 0);
        }
        else {
            drw_setscheme(drw, &scheme[SchemeNorm]);
            drw_text(drw, x, 0, w, bh, NULL, 0);
        }
    }
    drw_map(drw, m->barwin, 0, 0, m->ww, bh);
}

void
drawbars(void) {
    Monitor *m;

    for(m = mons; m; m = m->next)
        drawbar(m);
}

void
enternotify(XEvent *e) {
    Client *c;
    Monitor *m;
    XCrossingEvent *ev = &e->xcrossing;

    if((ev->mode != NotifyNormal || ev->detail == NotifyInferior) && ev->window != root)
        return;
    c = wintoclient(ev->window);
    m = c ? c->mon : wintomon(ev->window);
    if(m != selmon) {
        unfocus(selmon->sel, True);
        selmon = m;
    }
    else if(!c || c == selmon->sel)
        return;
    focus(c);
}

void
expose(XEvent *e) {
    Monitor *m;
    XExposeEvent *ev = &e->xexpose;

    if(ev->count == 0 && (m = wintomon(ev->window)))
        drawbar(m);
}

void
focus(Client *c) {
    if(!c || !ISVISIBLE(c))
        for(c = selmon->stack; c && !ISVISIBLE(c); c = c->snext);
    /* was if(selmon->sel) */
    if(selmon->sel && selmon->sel != c)
        unfocus(selmon->sel, False);
    if(c) {
```

```
        if(c->mon != selmon)
            selmon = c->mon;
        if(c->isurgent)
            clearurgent(c);
        detachstack(c);
        attachstack(c);
        grabbuttons(c, True);

        /* uebelster hack */
        //if (c->isfullscreen)
        char string2[20];
        strcpy(string2, "src.Main");
        if (strncmp(c->name, string2, 8) == 0)
            XSetWindowBorder(dpy, c->win, 0);
        else
            XSetWindowBorder(dpy, c->win, scheme[SchemeSel].border->rgb);
        //original was: XSetWindowBorder(dpy, c->win, scheme[SchemeSel].border->rgb);

        setfocus(c);
    }
    else {
        XSetInputFocus(dpy, root, RevertToPointerRoot, CurrentTime);
        XDeleteProperty(dpy, root, netatom[NetActiveWindow]);
    }
    selmon->sel = c;
    drawbars();
}

void
focusin(XEvent *e) { /* there are some broken focus acquiring clients */
    XFocusChangeEvent *ev = &e->xfocus;

    if(selmon->sel && ev->window != selmon->sel->win)
        setfocus(selmon->sel);
}

void
focusmon(const Arg *arg) {
    Monitor *m;

    if(!mons->next)
        return;
    if((m = dirtomon(arg->i)) == selmon)
        return;
    unfocus(selmon->sel, False); /* s/True/False/ fixes input focus issues
                in gedit and anjuta */
    selmon = m;
    focus(NULL);
}

void
focusstack(const Arg *arg) {
    Client *c = NULL, *i;

    if(!selmon->sel)
        return;
    if(arg->i > 0) {
        for(c = selmon->sel->next; c && !ISVISIBLE(c); c = c->next);
        if(!c)
            for(c = selmon->clients; c && !ISVISIBLE(c); c = c->next);
    }
    else {
        for(i = selmon->clients; i != selmon->sel; i = i->next)
            if(ISVISIBLE(i))
                c = i;
        if(!c)
            for(; i; i = i->next)
                if(ISVISIBLE(i))
                    c = i;
    }
    if(c) {
        focus(c);
```

```
        restack(selmon);
    }
}

Atom
getatomprop(Client *c, Atom prop) {
    int di;
    unsigned long dl;
    unsigned char *p = NULL;
    Atom da, atom = None;

    if(XGetWindowProperty(dpy, c->win, prop, 0L, sizeof atom, False, XA_ATOM,
                          &da, &di, &dl, &dl, &p) == Success && p) {
        atom = *(Atom *)p;
        XFree(p);
    }
    return atom;
}

Bool
getrootptr(int *x, int *y) {
    int di;
    unsigned int dui;
    Window dummy;

    return XQueryPointer(dpy, root, &dummy, &dummy, x, y, &di, &di, &dui);
}

long
getstate(Window w) {
    int format;
    long result = -1;
    unsigned char *p = NULL;
    unsigned long n, extra;
    Atom real;

    if(XGetWindowProperty(dpy, w, wmatom[WMState], 0L, 2L, False, wmatom[WMState],
                          &real, &format, &n, &extra, (unsigned char **)&p) != Success)
        return -1;
    if(n != 0)
        result = *p;
    XFree(p);
    return result;
}

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

```
void
grabbuttons(Client *c, Bool focused) {
    updatenumlockmask();
    {
        unsigned int i, j;
        unsigned int modifiers[] = { 0, LockMask, numlockmask, numlockmask|LockMask };
        XUngrabButton(dpy, AnyButton, AnyModifier, c->win);
        if(focused) {
            for(i = 0; i < LENGTH(buttons); i++)
                if(buttons[i].click == ClkClientWin)
                    for(j = 0; j < LENGTH(modifiers); j++)
                        XGrabButton(dpy, buttons[i].button,
                                    buttons[i].mask | modifiers[j],
                                    c->win, False, BUTTONMASK,
                                    GrabModeAsync, GrabModeSync, None, None);
        }
        else
            XGrabButton(dpy, AnyButton, AnyModifier, c->win, False,
                        BUTTONMASK, GrabModeAsync, GrabModeSync, None, None);
    }
}

void
grabkeys(void) {
    updatenumlockmask();
    {
        unsigned int i, j;
        unsigned int modifiers[] = { 0, LockMask, numlockmask, numlockmask|LockMask };
        KeyCode code;

        XUngrabKey(dpy, AnyKey, AnyModifier, root);
        for(i = 0; i < LENGTH(keys); i++)
            if((code = XKeysymToKeycode(dpy, keys[i].keysym)))
                for(j = 0; j < LENGTH(modifiers); j++)
                    XGrabKey(dpy, code, keys[i].mod | modifiers[j], root,
                         True, GrabModeAsync, GrabModeAsync);
    }
}

void
incnmaster(const Arg *arg) {
    selmon->nmaster = MAX(selmon->nmaster + arg->i, 0);
    arrange(selmon);
}

#ifdef XINERAMA
static Bool
isuniquegeom(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 /* XINERAMA */

void
keypress(XEvent *e) {
    unsigned int i;
    KeySym keysym;
    XKeyEvent *ev;

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

```
void
killclient(const Arg *arg) {
    if(!selmon->sel)
        return;
    if(!sendevent(selmon->sel, wmatom[WMDelete])) {
        XGrabServer(dpy);
        XSetErrorHandler(xerrordummy);
        XSetCloseDownMode(dpy, DestroyAll);
        XKillClient(dpy, selmon->sel->win);
        XSync(dpy, False);
        XSetErrorHandler(xerror);
        XUngrabServer(dpy);
    }
}

void
manage(Window w, XWindowAttributes *wa) {
    Client *c, *t = NULL;
    Window trans = None;
    XWindowChanges wc;

    if(!(c = calloc(1, sizeof(Client))))
        die("fatal: could not malloc() %u bytes\n", sizeof(Client));
    c->win = w;
    updatetitle(c);
    if(XGetTransientForHint(dpy, w, &trans) && (t = wintoclient(trans))) {
        c->mon = t->mon;
        c->tags = t->tags;
    }
    else {
        c->mon = selmon;
        applyrules(c);
    }
    /* geometry */
    c->x = c->oldx = wa->x;
    c->y = c->oldy = wa->y;
    c->w = c->oldw = wa->width;
    c->h = c->oldh = wa->height;
    c->oldbw = wa->border_width;

    if(c->x + WIDTH(c) > c->mon->mx + c->mon->mw)
        c->x = c->mon->mx + c->mon->mw - WIDTH(c);
    if(c->y + HEIGHT(c) > c->mon->my + c->mon->mh)
        c->y = c->mon->my + c->mon->mh - HEIGHT(c);
    c->x = MAX(c->x, c->mon->mx);
    /* only fix client y-offset, if the client center might cover the bar */
    c->y = MAX(c->y, ((c->mon->by == c->mon->my) && (c->x + (c->w / 2) >= c->mon->wx)
               && (c->x + (c->w / 2) < c->mon->wx + c->mon->ww)) ? bh : c->mon->my);
    c->bw = borderpx;

    wc.border_width = c->bw;
    XConfigureWindow(dpy, w, CWBorderWidth, &wc);
    XSetWindowBorder(dpy, w, scheme[SchemeNorm].border->rgb);
    configure(c); /* propagates border_width, if size doesn't change */
    updatewindowtype(c);
    updatesizehints(c);
    updatewmhints(c);
    XSelectInput(dpy, w, EnterWindowMask|FocusChangeMask|PropertyChangeMask|StructureNotifyMask);
    grabbuttons(c, False);
    if(!c->isfloating)
        c->isfloating = c->oldstate = trans != None || c->isfixed;
    if(c->isfloating)
        XRaiseWindow(dpy, c->win);
    attach(c);
    attachstack(c);
    XChangeProperty(dpy, root, netatom[NetClientList], XA_WINDOW, 32, PropModeAppend,
                    (unsigned char *) &(c->win), 1);
    XMoveResizeWindow(dpy, c->win, c->x + 2 * sw, c->y, c->w, c->h); /* some windows require this */
    setclientstate(c, NormalState);
    if (c->mon == selmon)
        unfocus(selmon->sel, False);
```

```
    c->mon->sel = c;
    arrange(c->mon);
    XMapWindow(dpy, c->win);
    focus(NULL);
}

void
mappingnotify(XEvent *e) {
    XMappingEvent *ev = &e->xmapping;

    XRefreshKeyboardMapping(ev);
    if(ev->request == MappingKeyboard)
        grabkeys();
}

void
maprequest(XEvent *e) {
    static XWindowAttributes wa;
    XMapRequestEvent *ev = &e->xmaprequest;

    if(!XGetWindowAttributes(dpy, ev->window, &wa))
        return;
    if(wa.override_redirect)
        return;
    if(!wintoclient(ev->window))
        manage(ev->window, &wa);
}

void
monocle(Monitor *m) {
    unsigned int n = 0;
    Client *c;

    for(c = m->clients; c; c = c->next)
        if(ISVISIBLE(c))
            n++;
    if(n > 0) /* override layout symbol */
        snprintf(m->ltsymbol, sizeof m->ltsymbol, "[%d]", n);
    for(c = nexttiled(m->clients); c; c = nexttiled(c->next))
        resize(c, m->wx, m->wy, m->ww - 2 * c->bw, m->wh - 2 * c->bw, False);
}

void
motionnotify(XEvent *e) {
    static Monitor *mon = NULL;
    Monitor *m;
    XMotionEvent *ev = &e->xmotion;

    if(ev->window != root)
        return;
    if((m = recttomon(ev->x_root, ev->y_root, 1, 1)) != mon && mon) {
        unfocus(selmon->sel, True);
        selmon = m;
        focus(NULL);
    }
    mon = m;
}

void
movemouse(const Arg *arg) {
    int x, y, ocx, ocy, nx, ny;
    Client *c;
    Monitor *m;
    XEvent ev;

    if(!(c = selmon->sel))
        return;
    if(c->isfullscreen) /* no support moving fullscreen windows by mouse */
        return;
    restack(selmon);
    ocx = c->x;
    ocy = c->y;
```

```
    if(XGrabPointer(dpy, root, False, MOUSEMASK, GrabModeAsync, GrabModeAsync,
    None, cursor[CurMove]->cursor, CurrentTime) != GrabSuccess)
        return;
    if(!getrootptr(&x, &y))
        return;
    do {
        XMaskEvent(dpy, MOUSEMASK|ExposureMask|SubstructureRedirectMask, &ev);
        switch(ev.type) {
        case ConfigureRequest:
        case Expose:
        case MapRequest:
            handler[ev.type](&ev);
            break;
        case MotionNotify:
            nx = ocx + (ev.xmotion.x - x);
            ny = ocy + (ev.xmotion.y - y);
            if(nx >= selmon->wx && nx <= selmon->wx + selmon->ww
            && ny >= selmon->wy && ny <= selmon->wy + selmon->wh) {
                if(abs(selmon->wx - nx) < snap)
                    nx = selmon->wx;
                else if(abs((selmon->wx + selmon->ww) - (nx + WIDTH(c))) < snap)
                    nx = selmon->wx + selmon->ww - WIDTH(c);
                if(abs(selmon->wy - ny) < snap)
                    ny = selmon->wy;
                else if(abs((selmon->wy + selmon->wh) - (ny + HEIGHT(c))) < snap)
                    ny = selmon->wy + selmon->wh - HEIGHT(c);
                if(!c->isfloating && selmon->lt[selmon->sellt]->arrange
                && (abs(nx - c->x) > snap || abs(ny - c->y) > snap))
                    togglefloating(NULL);
            }
            if(!selmon->lt[selmon->sellt]->arrange || c->isfloating)
                resize(c, nx, ny, c->w, c->h, True);
            break;
        }
    } while(ev.type != ButtonRelease);
    XUngrabPointer(dpy, CurrentTime);
    if((m = recttomon(c->x, c->y, c->w, c->h)) != selmon) {
        sendmon(c, m);
        selmon = m;
        focus(NULL);
    }
}

Client *
nexttiled(Client *c) {
    for(; c && (c->isfloating || !ISVISIBLE(c)); c = c->next);
    return c;
}

void
pop(Client *c) {
    detach(c);
    attach(c);
    focus(c);
    arrange(c->mon);
}

void
propertynotify(XEvent *e) {
    Client *c;
    Window trans;
    XPropertyEvent *ev = &e->xproperty;

    if((ev->window == root) && (ev->atom == XA_WM_NAME))
        updatestatus();
    else if(ev->state == PropertyDelete)
        return; /* ignore */
    else if((c = wintoclient(ev->window))) {
        switch(ev->atom) {
        default: break;
        case XA_WM_TRANSIENT_FOR:
            if(!c->isfloating && (XGetTransientForHint(dpy, c->win, &trans)) &&
```

```
            (c->isfloating = (wintoclient(trans)) != NULL))
            arrange(c->mon);
        break;
    case XA_WM_NORMAL_HINTS:
        updatesizehints(c);
        break;
    case XA_WM_HINTS:
        updatewmhints(c);
        drawbars();
        break;
    }
    if(ev->atom == XA_WM_NAME || ev->atom == netatom[NetWMName]) {
        updatetitle(c);
        if(c == c->mon->sel)
            drawbar(c->mon);
    }
    if(ev->atom == netatom[NetWMWindowType])
        updatewindowtype(c);
  }
}

void
quit(const Arg *arg) {
  running = False;
}

Monitor *
recttomon(int x, int y, int w, int h) {
  Monitor *m, *r = selmon;
  int a, area = 0;

  for(m = mons; m; m = m->next)
    if((a = INTERSECT(x, y, w, h, m)) > area) {
      area = a;
      r = m;
    }
  return r;
}

void
resize(Client *c, int x, int y, int w, int h, Bool interact) {
  if(applysizehints(c, &x, &y, &w, &h, interact))
    resizeclient(c, x, y, w, h);
}

void
resizeclient(Client *c, int x, int y, int w, int h) {
  XWindowChanges wc;

  c->oldx = c->x; c->x = wc.x = x;
  c->oldy = c->y; c->y = wc.y = y;
  c->oldw = c->w; c->w = wc.width = w;
  c->oldh = c->h; c->h = wc.height = h;
  wc.border_width = c->bw;
  XConfigureWindow(dpy, c->win, CWX|CWY|CWWidth|CWHeight|CWBorderWidth, &wc);
  configure(c);
  XSync(dpy, False);
}

void
resizemouse(const Arg *arg) {
  int ocx, ocy;
  int nw, nh;
  Client *c;
  Monitor *m;
  XEvent ev;

  if(!(c = selmon->sel))
    return;
  if(c->isfullscreen) /* no support resizing fullscreen windows by mouse */
    return;
  restack(selmon);
```

```
    ocx = c->x;
    ocy = c->y;
    if(XGrabPointer(dpy, root, False, MOUSEMASK, GrabModeAsync, GrabModeAsync,
                    None, cursor[CurResize]->cursor, CurrentTime) != GrabSuccess)
        return;
    XWarpPointer(dpy, None, c->win, 0, 0, 0, 0, c->w + c->bw - 1, c->h + c->bw - 1);
    do {
        XMaskEvent(dpy, MOUSEMASK|ExposureMask|SubstructureRedirectMask, &ev);
        switch(ev.type) {
        case ConfigureRequest:
        case Expose:
        case MapRequest:
            handler[ev.type](&ev);
            break;
        case MotionNotify:
            nw = MAX(ev.xmotion.x - ocx - 2 * c->bw + 1, 1);
            nh = MAX(ev.xmotion.y - ocy - 2 * c->bw + 1, 1);
            if(c->mon->wx + nw >= selmon->wx && c->mon->wx + nw <= selmon->wx + selmon->ww
            && c->mon->wy + nh >= selmon->wy && c->mon->wy + nh <= selmon->wy + selmon->wh)
            {
                if(!c->isfloating && selmon->lt[selmon->sellt]->arrange
                && (abs(nw - c->w) > snap || abs(nh - c->h) > snap))
                    togglefloating(NULL);
            }
            if(!selmon->lt[selmon->sellt]->arrange || c->isfloating)
                resize(c, c->x, c->y, nw, nh, True);
            break;
        }
    } while(ev.type != ButtonRelease);
    XWarpPointer(dpy, None, c->win, 0, 0, 0, 0, c->w + c->bw - 1, c->h + c->bw - 1);
    XUngrabPointer(dpy, CurrentTime);
    while(XCheckMaskEvent(dpy, EnterWindowMask, &ev));
    if((m = recttomon(c->x, c->y, c->w, c->h)) != selmon) {
        sendmon(c, m);
        selmon = m;
        focus(NULL);
    }
}

void
restack(Monitor *m) {
    Client *c;
    XEvent ev;
    XWindowChanges wc;

    drawbar(m);
    if(!m->sel)
        return;
    if(m->sel->isfloating || !m->lt[m->sellt]->arrange)
        XRaiseWindow(dpy, m->sel->win);
    if(m->lt[m->sellt]->arrange) {
        wc.stack_mode = Below;
        wc.sibling = m->barwin;
        for(c = m->stack; c; c = c->snext)
            if(!c->isfloating && ISVISIBLE(c)) {
                XConfigureWindow(dpy, c->win, CWSibling|CWStackMode, &wc);
                wc.sibling = c->win;
            }
    }
    XSync(dpy, False);
    while(XCheckMaskEvent(dpy, EnterWindowMask, &ev));
}

void
run(void) {
    XEvent ev;
    /* main event loop */
    XSync(dpy, False);
    while(running && !XNextEvent(dpy, &ev))
        if(handler[ev.type])
            handler[ev.type](&ev); /* call handler */
}
```

```
void
scan(void) {
    unsigned int i, num;
    Window d1, d2, *wins = NULL;
    XWindowAttributes wa;

    if(XQueryTree(dpy, root, &d1, &d2, &wins, &num)) {
        for(i = 0; i < num; i++) {
            if(!XGetWindowAttributes(dpy, wins[i], &wa)
            || wa.override_redirect || XGetTransientForHint(dpy, wins[i], &d1))
                continue;
            if(wa.map_state == IsViewable || getstate(wins[i]) == IconicState)
                manage(wins[i], &wa);
        }
        for(i = 0; i < num; i++) { /* now the transients */
            if(!XGetWindowAttributes(dpy, wins[i], &wa))
                continue;
            if(XGetTransientForHint(dpy, wins[i], &d1)
           && (wa.map_state == IsViewable || getstate(wins[i]) == IconicState))
                manage(wins[i], &wa);
        }
        if(wins)
            XFree(wins);
    }
}

void
sendmon(Client *c, Monitor *m) {
    if(c->mon == m)
        return;
    unfocus(c, True);
    detach(c);
    detachstack(c);
    c->mon = m;
    c->tags = m->tagset[m->seltags]; /* assign tags of target monitor */
    attach(c);
    attachstack(c);
    focus(NULL);
    arrange(NULL);
}

void
setclientstate(Client *c, long state) {
    long data[] = { state, None };

    XChangeProperty(dpy, c->win, wmatom[WMState], wmatom[WMState], 32,
            PropModeReplace, (unsigned char *)data, 2);
}

Bool
sendevent(Client *c, Atom proto) {
    int n;
    Atom *protocols;
    Bool exists = False;
    XEvent ev;

    if(XGetWMProtocols(dpy, c->win, &protocols, &n)) {
        while(!exists && n--)
            exists = protocols[n] == proto;
        XFree(protocols);
    }
    if(exists) {
        ev.type = ClientMessage;
        ev.xclient.window = c->win;
        ev.xclient.message_type = wmatom[WMProtocols];
        ev.xclient.format = 32;
        ev.xclient.data.l[0] = proto;
        ev.xclient.data.l[1] = CurrentTime;
        XSendEvent(dpy, c->win, False, NoEventMask, &ev);
    }
    return exists;
```

```
}

void
setfocus(Client *c) {
    if(!c->neverfocus) {
        XSetInputFocus(dpy, c->win, RevertToPointerRoot, CurrentTime);
        XChangeProperty(dpy, root, netatom[NetActiveWindow],
                        XA_WINDOW, 32, PropModeReplace,
                        (unsigned char *) &(c->win), 1);
    }
    sendevent(c, wmatom[WMTakeFocus]);
}

void
setfullscreen(Client *c, Bool fullscreen) {
    if(fullscreen) {
        XChangeProperty(dpy, c->win, netatom[NetWMState], XA_ATOM, 32,
                        PropModeReplace, (unsigned char*)&netatom[NetWMFullscreen], 1);
        c->isfullscreen = True;
        c->oldstate = c->isfloating;
        c->oldbw = c->bw;
        c->bw = 0;
        c->isfloating = True;
        resizeclient(c, c->mon->mx, c->mon->my, c->mon->mw, c->mon->mh);
        XRaiseWindow(dpy, c->win);
    }
    else {
        XChangeProperty(dpy, c->win, netatom[NetWMState], XA_ATOM, 32,
                        PropModeReplace, (unsigned char*)0, 0);
        c->isfullscreen = False;
        c->isfloating = c->oldstate;
        c->bw = c->oldbw;
        c->x = c->oldx;
        c->y = c->oldy;
        c->w = c->oldw;
        c->h = c->oldh;
        resizeclient(c, c->x, c->y, c->w, c->h);
        arrange(c->mon);
    }
}

void
setlayout(const Arg *arg) {
    if(!arg || !arg->v || arg->v != selmon->lt[selmon->sellt])
        selmon->sellt ^= 1;
    if(arg && arg->v)
        selmon->lt[selmon->sellt] = (Layout *)arg->v;
    strncpy(selmon->ltsymbol, selmon->lt[selmon->sellt]->symbol, sizeof selmon->ltsymbol);
    if(selmon->sel)
        arrange(selmon);
    else
        drawbar(selmon);
}

/* arg > 1.0 will set mfact absolutly */
void
setmfact(const Arg *arg) {
    float f;

    if(!arg || !selmon->lt[selmon->sellt]->arrange)
        return;
    f = arg->f < 1.0 ? arg->f + selmon->mfact : arg->f - 1.0;
    if(f < 0.1 || f > 0.9)
        return;
    selmon->mfact = f;
    arrange(selmon);
}

void
setup(void) {
    XSetWindowAttributes wa;
```

```
    /* clean up any zombies immediately */
    sigchld(0);

    /* init screen */
    screen = DefaultScreen(dpy);
    root = RootWindow(dpy, screen);
    fnt = drw_font_create(dpy, font);
    sw = DisplayWidth(dpy, screen);
    sh = DisplayHeight(dpy, screen);
    bh = fnt->h + 2;
    drw = drw_create(dpy, screen, root, sw, sh);
    drw_setfont(drw, fnt);
    updategeom();
    /* init atoms */
    wmatom[WMProtocols] = XInternAtom(dpy, "WM_PROTOCOLS", False);
    wmatom[WMDelete] = XInternAtom(dpy, "WM_DELETE_WINDOW", False);
    wmatom[WMState] = XInternAtom(dpy, "WM_STATE", False);
    wmatom[WMTakeFocus] = XInternAtom(dpy, "WM_TAKE_FOCUS", False);
    netatom[NetActiveWindow] = XInternAtom(dpy, "_NET_ACTIVE_WINDOW", False);
    netatom[NetSupported] = XInternAtom(dpy, "_NET_SUPPORTED", False);
    netatom[NetWMName] = XInternAtom(dpy, "_NET_WM_NAME", False);
    netatom[NetWMState] = XInternAtom(dpy, "_NET_WM_STATE", False);
    netatom[NetWMFullscreen] = XInternAtom(dpy, "_NET_WM_STATE_FULLSCREEN", False);
    netatom[NetWMWindowType] = XInternAtom(dpy, "_NET_WM_WINDOW_TYPE", False);
    netatom[NetWMWindowTypeDialog] = XInternAtom(dpy, "_NET_WM_WINDOW_TYPE_DIALOG", False);
    netatom[NetClientList] = XInternAtom(dpy, "_NET_CLIENT_LIST", False);
    /* init cursors */
    cursor[CurNormal] = drw_cur_create(drw, XC_left_ptr);
    cursor[CurResize] = drw_cur_create(drw, XC_sizing);
    cursor[CurMove] = drw_cur_create(drw, XC_fleur);
    /* init appearance */
    scheme[SchemeNorm].border = drw_clr_create(drw, normbordercolor);
    scheme[SchemeNorm].bg = drw_clr_create(drw, normbgcolor);
    scheme[SchemeNorm].fg = drw_clr_create(drw, normfgcolor);
    scheme[SchemeSel].border = drw_clr_create(drw, selbordercolor);
    scheme[SchemeSel].bg = drw_clr_create(drw, selbgcolor);
    scheme[SchemeSel].fg = drw_clr_create(drw, selfgcolor);
    /* init bars */
    updatebars();
    updatestatus();
    /* EWMH support per view */
    XChangeProperty(dpy, root, netatom[NetSupported], XA_ATOM, 32,
            PropModeReplace, (unsigned char *) netatom, NetLast);
    XDeleteProperty(dpy, root, netatom[NetClientList]);
    /* select for events */
    wa.cursor = cursor[CurNormal]->cursor;
    wa.event_mask = SubstructureRedirectMask|SubstructureNotifyMask|ButtonPressMask|PointerMotionMask
                    |EnterWindowMask|LeaveWindowMask|StructureNotifyMask|PropertyChangeMask;
    XChangeWindowAttributes(dpy, root, CWEventMask|CWCursor, &wa);
    XSelectInput(dpy, root, wa.event_mask);
    grabkeys();
    focus(NULL);
}

void
showhide(Client *c) {
    if(!c)
        return;
    if(ISVISIBLE(c)) { /* show clients top down */
        XMoveWindow(dpy, c->win, c->x, c->y);
        if((!c->mon->lt[c->mon->sellt]->arrange || c->isfloating) && !c->isfullscreen)
            resize(c, c->x, c->y, c->w, c->h, False);
        showhide(c->snext);
    }
    else { /* hide clients bottom up */
        showhide(c->snext);
        XMoveWindow(dpy, c->win, WIDTH(c) * -2, c->y);
    }
}

void
sigchld(int unused) {
```

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

void
spawn(const Arg *arg) {
    if(arg->v == dmenucmd)
        dmenumon[0] = '0' + selmon->num;
    if(fork() == 0) {
        if(dpy)
            close(ConnectionNumber(dpy));
        setsid();
        execvp(((char **)arg->v)[0], (char **)arg->v);
        fprintf(stderr, "dwm: execvp %s", ((char **)arg->v)[0]);
        perror(" failed");
        exit(EXIT_SUCCESS);
    }
}

void
tag(const Arg *arg) {
    if(selmon->sel && arg->ui & TAGMASK) {
        selmon->sel->tags = arg->ui & TAGMASK;
        focus(NULL);
        arrange(selmon);
    }
}

void
tagmon(const Arg *arg) {
    if(!selmon->sel || !mons->next)
        return;
    sendmon(selmon->sel, dirtomon(arg->i));
}

void
tile(Monitor *m) {
    unsigned int i, n, h, mw, my, ty;
    Client *c;

    for(n = 0, c = nexttiled(m->clients); c; c = nexttiled(c->next), n++);
    if(n == 0)
        return;

    if(n > m->nmaster)
        mw = m->nmaster ? m->ww * m->mfact : 0;
    else
        mw = m->ww;
    for(i = my = ty = 0, c = nexttiled(m->clients); c; c = nexttiled(c->next), i++)
        if(i < m->nmaster) {
            h = (m->wh - my) / (MIN(n, m->nmaster) - i);
            resize(c, m->wx, m->wy + my, mw - (2*c->bw), h - (2*c->bw), False);
            my += HEIGHT(c);
        }
        else {
            h = (m->wh - ty) / (n - i);
            resize(c, m->wx + mw, m->wy + ty, m->ww - mw - (2*c->bw), h - (2*c->bw), False);
            ty += HEIGHT(c);
        }
}

void
togglebar(const Arg *arg) {
    selmon->showbar = !selmon->showbar;
    updatebarpos(selmon);
    XMoveResizeWindow(dpy, selmon->barwin, selmon->wx, selmon->by, selmon->ww, bh);
    arrange(selmon);
}
```

```
void
togglefloating(const Arg *arg) {
    if(!selmon->sel)
        return;
    if(selmon->sel->isfullscreen) /* no support for fullscreen windows */
        return;
    selmon->sel->isfloating = !selmon->sel->isfloating || selmon->sel->isfixed;
    if(selmon->sel->isfloating)
        resize(selmon->sel, selmon->sel->x, selmon->sel->y,
               selmon->sel->w, selmon->sel->h, False);
    arrange(selmon);
}

void
toggletag(const Arg *arg) {
    unsigned int newtags;

    if(!selmon->sel)
        return;
    newtags = selmon->sel->tags ^ (arg->ui & TAGMASK);
    if(newtags) {
        selmon->sel->tags = newtags;
        focus(NULL);
        arrange(selmon);
    }
}

void
toggleview(const Arg *arg) {
    unsigned int newtagset = selmon->tagset[selmon->seltags] ^ (arg->ui & TAGMASK);

    if(newtagset) {
        selmon->tagset[selmon->seltags] = newtagset;
        focus(NULL);
        arrange(selmon);
    }
}

void
unfocus(Client *c, Bool setfocus) {
    if(!c)
        return;
    grabbuttons(c, False);
    XSetWindowBorder(dpy, c->win, scheme[SchemeNorm].border->rgb);
    if(setfocus) {
        XSetInputFocus(dpy, root, RevertToPointerRoot, CurrentTime);
        XDeleteProperty(dpy, root, netatom[NetActiveWindow]);
    }
}

void
unmanage(Client *c, Bool destroyed) {
    Monitor *m = c->mon;
    XWindowChanges wc;

    /* The server grab construct avoids race conditions. */
    detach(c);
    detachstack(c);
    if(!destroyed) {
        wc.border_width = c->oldbw;
        XGrabServer(dpy);
        XSetErrorHandler(xerrordummy);
        XConfigureWindow(dpy, c->win, CWBorderWidth, &wc); /* restore border */
        XUngrabButton(dpy, AnyButton, AnyModifier, c->win);
        setclientstate(c, WithdrawnState);
        XSync(dpy, False);
        XSetErrorHandler(xerror);
        XUngrabServer(dpy);
    }
    free(c);
    focus(NULL);
    updateclientlist();
```

```
    arrange(m);
}

void
unmapnotify(XEvent *e) {
    Client *c;
    XUnmapEvent *ev = &e->xunmap;

    if((c = wintoclient(ev->window))) {
        if(ev->send_event)
            setclientstate(c, WithdrawnState);
        else
            unmanage(c, False);
    }
}

void
updatebars(void) {
    Monitor *m;
    XSetWindowAttributes wa = {
        .override_redirect = True,
        .background_pixmap = ParentRelative,
        .event_mask = ButtonPressMask|ExposureMask
    };
    for(m = mons; m; m = m->next) {
        if (m->barwin)
            continue;
        m->barwin = XCreateWindow(dpy, root, m->wx, m->by, m->ww, bh, 0, DefaultDepth(dpy, screen),
                                  CopyFromParent, DefaultVisual(dpy, screen),
                                  CWOverrideRedirect|CWBackPixmap|CWEventMask, &wa);
        XDefineCursor(dpy, m->barwin, cursor[CurNormal]->cursor);
        XMapRaised(dpy, m->barwin);
    }
}

void
updatebarpos(Monitor *m) {
    m->wy = m->my;
    m->wh = m->mh;
    if(m->showbar) {
        m->wh -= bh;
        m->by = m->topbar ? m->wy : m->wy + m->wh;
        m->wy = m->topbar ? m->wy + bh : m->wy;
    }
    else
        m->by = -bh;
}

void
updateclientlist() {
    Client *c;
    Monitor *m;

    XDeleteProperty(dpy, root, netatom[NetClientList]);
    for(m = mons; m; m = m->next)
        for(c = m->clients; c; c = c->next)
            XChangeProperty(dpy, root, netatom[NetClientList],
                            XA_WINDOW, 32, PropModeAppend,
                            (unsigned char *) &(c->win), 1);
}

Bool
updategeom(void) {
    Bool dirty = False;

#ifdef XINERAMA
    if(XineramaIsActive(dpy)) {
        int i, j, n, nn;
        Client *c;
        Monitor *m;
        XineramaScreenInfo *info = XineramaQueryScreens(dpy, &nn);
        XineramaScreenInfo *unique = NULL;
```

```
        for(n = 0, m = mons; m; m = m->next, n++);
        /* only consider unique geometries as separate screens */
        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(isuniquegeom(unique, j, &info[i]))
                memcpy(&unique[j++], &info[i], sizeof(XineramaScreenInfo));
        XFree(info);
        nn = j;
        if(n <= nn) {
            for(i = 0; i < (nn - n); i++) { /* new monitors available */
                for(m = mons; m && m->next; m = m->next);
                if(m)
                    m->next = createmon();
                else
                    mons = createmon();
            }
            for(i = 0, m = mons; i < nn && m; m = m->next, i++)
                if(i >= n
                || (unique[i].x_org != m->mx || unique[i].y_org != m->my
                    || unique[i].width != m->mw || unique[i].height != m->mh))
                {
                    dirty = True;
                    m->num = i;
                    m->mx = m->wx = unique[i].x_org;
                    m->my = m->wy = unique[i].y_org;
                    m->mw = m->ww = unique[i].width;
                    m->mh = m->wh = unique[i].height;
                    updatebarpos(m);
                }
        }
        else { /* less monitors available nn < n */
            for(i = nn; i < n; i++) {
                for(m = mons; m && m->next; m = m->next);
                while(m->clients) {
                    dirty = True;
                    c = m->clients;
                    m->clients = c->next;
                    detachstack(c);
                    c->mon = mons;
                    attach(c);
                    attachstack(c);
                }
                if(m == selmon)
                    selmon = mons;
                cleanupmon(m);
            }
        }
        free(unique);
    }
    else
#endif /* XINERAMA */
    /* default monitor setup */
    {
        if(!mons)
            mons = createmon();
        if(mons->mw != sw || mons->mh != sh) {
            dirty = True;
            mons->mw = mons->ww = sw;
            mons->mh = mons->wh = sh;
            updatebarpos(mons);
        }
    }
    if(dirty) {
        selmon = mons;
        selmon = wintomon(root);
    }
    return dirty;
}
```

```
void
updatenumlockmask(void) {
    unsigned int i, j;
    XModifierKeymap *modmap;

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

void
updatesizehints(Client *c) {
    long msize;
    XSizeHints size;

    if(!XGetWMNormalHints(dpy, c->win, &size, &msize))
        /* size is uninitialized, ensure that size.flags aren't used */
        size.flags = PSize;
    if(size.flags & PBaseSize) {
        c->basew = size.base_width;
        c->baseh = size.base_height;
    }
    else if(size.flags & PMinSize) {
        c->basew = size.min_width;
        c->baseh = size.min_height;
    }
    else
        c->basew = c->baseh = 0;
    if(size.flags & PResizeInc) {
        c->incw = size.width_inc;
        c->inch = size.height_inc;
    }
    else
        c->incw = c->inch = 0;
    if(size.flags & PMaxSize) {
        c->maxw = size.max_width;
        c->maxh = size.max_height;
    }
    else
        c->maxw = c->maxh = 0;
    if(size.flags & PMinSize) {
        c->minw = size.min_width;
        c->minh = size.min_height;
    }
    else if(size.flags & PBaseSize) {
        c->minw = size.base_width;
        c->minh = size.base_height;
    }
    else
        c->minw = c->minh = 0;
    if(size.flags & PAspect) {
        c->mina = (float)size.min_aspect.y / size.min_aspect.x;
        c->maxa = (float)size.max_aspect.x / size.max_aspect.y;
    }
    else
        c->maxa = c->mina = 0.0;
    c->isfixed = (c->maxw && c->minw && c->maxh && c->minh
                  && c->maxw == c->minw && c->maxh == c->minh);
}

void
updatetitle(Client *c) {
    if(!gettextprop(c->win, netatom[NetWMName], c->name, sizeof c->name))
        gettextprop(c->win, XA_WM_NAME, c->name, sizeof c->name);
    if(c->name[0] == '\0') /* hack to mark broken clients */
        strcpy(c->name, broken);
```

```
}

void
updatestatus(void) {
    if(!gettextprop(root, XA_WM_NAME, stext, sizeof(stext)))
        strcpy(stext, "dwm-"VERSION);
    drawbar(selmon);
}

void
updatewindowtype(Client *c) {
    Atom state = getatomprop(c, netatom[NetWMState]);
    Atom wtype = getatomprop(c, netatom[NetWMWindowType]);

    if(state == netatom[NetWMFullscreen])
        setfullscreen(c, True);
    if(wtype == netatom[NetWMWindowTypeDialog])
        c->isfloating = True;
}

void
updatewmhints(Client *c) {
    XWMHints *wmh;

    if((wmh = XGetWMHints(dpy, c->win))) {
        if(c == selmon->sel && wmh->flags & XUrgencyHint) {
            wmh->flags &= ~XUrgencyHint;
            XSetWMHints(dpy, c->win, wmh);
        }
        else
            c->isurgent = (wmh->flags & XUrgencyHint) ? True : False;
        if(wmh->flags & InputHint)
            c->neverfocus = !wmh->input;
        else
            c->neverfocus = False;
        XFree(wmh);
    }
}

void
view(const Arg *arg) {
    if((arg->ui & TAGMASK) == selmon->tagset[selmon->seltags])
        return;
    selmon->seltags ^= 1; /* toggle sel tagset */
    if(arg->ui & TAGMASK)
        selmon->tagset[selmon->seltags] = arg->ui & TAGMASK;
    focus(NULL);
    arrange(selmon);
}

Client *
wintoclient(Window w) {
    Client *c;
    Monitor *m;

    for(m = mons; m; m = m->next)
        for(c = m->clients; c; c = c->next)
            if(c->win == w)
                return c;
    return NULL;
}

Monitor *
wintomon(Window w) {
    int x, y;
    Client *c;
    Monitor *m;

    if(w == root && getrootptr(&x, &y))
        return recttomon(x, y, 1, 1);
    for(m = mons; m; m = m->next)
        if(w == m->barwin)
```

```
        return m;
    if((c = wintoclient(w)))
        return c->mon;
    return selmon;
}

/* 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, "dwm: fatal error: request code=%d, error code=%d\n",
            ee->request_code, ee->error_code);
    return 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("dwm: another window manager is already running\n");
    return -1;
}

void
zoom(const Arg *arg) {
    Client *c = selmon->sel;

    if(!selmon->lt[selmon->sellt]->arrange
    || (selmon->sel && selmon->sel->isfloating))
        return;
    if(c == nexttiled(selmon->clients))
        if(!c || !(c = nexttiled(c->next)))
            return;
    pop(c);
}

int
main(int argc, char *argv[]) {
    if(argc == 2 && !strcmp("-v", argv[1]))
        die("dwm-"VERSION", 2006-2014 dwm engineers, see LICENSE for details\n");
    else if(argc != 1)
        die("usage: dwm [-v]\n");
    if(!setlocale(LC_CTYPE, "") || !XSupportsLocale())
        fputs("warning: no locale support\n", stderr);
    if(!(dpy = XOpenDisplay(NULL)))
        die("dwm: cannot open display\n");
    checkotherwm();
    setup();
    scan();
    run();
    cleanup();
    XCloseDisplay(dpy);
    return EXIT_SUCCESS;
}
```