

```
/* ev-annotation-window.c
 * this file is part of evince, a gnome document viewer
 *
 * Copyright (C) 2009 Carlos Garcia Campos <carlosgc@gnome.org>
 * Copyright (C) 2007 Iñigo Martínez <inigomartinez@gmail.com>
 *
 * Evince is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * Evince is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301, USA.
 */

#include "config.h"

#include <string.h>

#include "ev-annotation-window.h"
#include "ev-stock-icons.h"
#include "ev-view-marshall.h"
#include "ev-document-misc.h"

enum {
    PROP_0,
    PROP_ANNOTATION,
    PROP_PARENT
};

enum {
    CLOSED,
    MOVED,
    N_SIGNALS
};

struct _EvAnnotationWindow {
    GtkWindow      base_instance;

    EvAnnotation *annotation;
    GtkWindow      *parent;

    GtkWidget      *title;
    GtkWidget      *close_button;
    GtkWidget      *text_view;
    GtkWidget      *resize_se;
    GtkWidget      *resize_sw;

    gboolean        is_open;
    EvRectangle     *rect;

    gboolean        in_move;
    gint            x;
    gint            y;
    gint            orig_x;
    gint            orig_y;
};

struct _EvAnnotationWindowClass {
    GtkWindowClass base_class;

    void (* closed) (EvAnnotationWindow *window);
    void (* moved) (EvAnnotationWindow *window,
                   gint x,
                   gint y);
};

static guint signals[N_SIGNALS];

G_DEFINE_TYPE (EvAnnotationWindow, ev_annotation_window, GTK_TYPE_WINDOW)

#define EV_ICON_SIZE_ANNOT_WINDOW (ev_annotation_window_get_icon_size())
```

```

/* Cut and paste from gtkwindow.c */
static void
send_focus_change (GtkWidget *widget,
                   gboolean in)
{
    GdkEvent *fevent = gdk_event_new (GDK_FOCUS_CHANGE);

    fevent->focus_change.type = GDK_FOCUS_CHANGE;
    fevent->focus_change.window = gtk_widget_get_window (widget);
    fevent->focus_change.in = in;
    if (fevent->focus_change.window)
        g_object_ref (fevent->focus_change.window);

    gtk_widget_send_focus_change (widget, fevent);

    gdk_event_free (fevent);
}

static gdouble
get_screen_dpi (EvAnnotationWindow *window)
{
    GdkScreen *screen;

    screen = gtk_window_get_screen (GTK_WINDOW (window));
    return ev_document_misc_get_screen_dpi (screen);
}

static GtkIconSize
ev_annotation_window_get_icon_size (void)
{
    static GtkIconSize icon_size = 0;

    if (G_UNLIKELY (icon_size == 0))
        icon_size = gtk_icon_size_register ("ev-icon-size-annot-window", 8, 8);

    return icon_size;
}

static void
ev_annotation_window_check_contents_modified (EvAnnotationWindow *window)
{
    gchar *contents;
    GtkTextIter start, end;
    GtkTextBuffer *buffer;
    EvAnnotation *annot = window->annotation;

    buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (window->text_view));
    gtk_text_buffer_get_bounds (buffer, &start, &end);
    contents = gtk_text_buffer_get_text (buffer, &start, &end, FALSE);

    if (contents && annot->contents) {
        if (strcmp (contents, annot->contents) != 0) {
            g_free (annot->contents);
            annot->contents = contents;
            annot->changed = TRUE;
        } else {
            g_free (contents);
        }
    } else if (annot->contents) {
        g_free (annot->contents);
        annot->contents = NULL;
        annot->changed = TRUE;
    } else if (contents) {
        annot->contents = contents;
        annot->changed = TRUE;
    }
}

static void
ev_annotation_window_set_color (EvAnnotationWindow *window,
                               GdkColor *color)
{
    GtkRcStyle *rc_style;
    GdkColor gcolor;

    gcolor = *color;

```

```

/* Allocate these colors */
gdk_colormap_alloc_color (gtk_widget_get_colormap (GTK_WIDGET (window)),
                          &gcolor, FALSE, TRUE);

/* Apply colors to style */
rc_style = gtk_widget_get_modifier_style (GTK_WIDGET (window));
rc_style->base[GTK_STATE_NORMAL] = gcolor;
rc_style->bg[GTK_STATE_PRELIGHT] = gcolor;
rc_style->bg[GTK_STATE_NORMAL] = gcolor;
rc_style->bg[GTK_STATE_ACTIVE] = gcolor;
rc_style->color_flags[GTK_STATE_PRELIGHT] = GTK_RC_BG;
rc_style->color_flags[GTK_STATE_NORMAL] = GTK_RC_BG | GTK_RC_BASE;
rc_style->color_flags[GTK_STATE_ACTIVE] = GTK_RC_BG;

/* Apply the style to the widgets */
g_object_ref (rc_style);
gtk_widget_modify_style (GTK_WIDGET (window), rc_style);
gtk_widget_modify_style (window->close_button, rc_style);
gtk_widget_modify_style (window->resize_se, rc_style);
gtk_widget_modify_style (window->resize_sw, rc_style);
g_object_unref (rc_style);
}

static void
ev_annotation_window_dispose (GObject *object)
{
    EvAnnotationWindow *window = EV_ANNOTATION_WINDOW (object);

    if (window->annotation) {
        ev_annotation_window_check_contents_modified (window);
        g_object_unref (window->annotation);
        window->annotation = NULL;
    }

    if (window->rect) {
        ev_rectangle_free (window->rect);
        window->rect = NULL;
    }

    (* G_OBJECT_CLASS (ev_annotation_window_parent_class)->dispose) (object);
}

static void
ev_annotation_window_set_property (GObject      *object,
                                   guint         prop_id,
                                   const GValue *value,
                                   GParamSpec   *pspec)
{
    EvAnnotationWindow *window = EV_ANNOTATION_WINDOW (object);

    switch (prop_id) {
        case PROP_ANNOTATION:
            window->annotation = g_value_dup_object (value);
            break;
        case PROP_PARENT:
            window->parent = g_value_get_object (value);
            break;
        default:
            G_OBJECT_WARN_INVALID_PROPERTY_ID (object, prop_id, pspec);
    }
}

static gboolean
ev_annotation_window_resize (EvAnnotationWindow *window,
                             GdkEventButton     *event,
                             GtkWidget           *ebox)
{
    if (event->type == GDK_BUTTON_PRESS && event->button == 1) {
        gtk_window_begin_resize_drag (GTK_WINDOW (window),
                                      window->resize_sw == ebox ?
                                      GDK_WINDOW_EDGE_SOUTH_WEST :
                                      GDK_WINDOW_EDGE_SOUTH_EAST,
                                      event->button, event->x_root,
                                      event->y_root, event->time);

        return TRUE;
    }

    return FALSE;
}

```

```
}

static void
ev_annotation_window_set_resize_cursor (GtkWidget      *widget,
                                         EvAnnotationWindow *window)
{
    GdkWindow *gdk_window = gtk_widget_get_window (widget);

    if (!gdk_window)
        return;

    if (gtk_widget_is_sensitive (widget)) {
        GdkDisplay *display = gtk_widget_get_display (widget);
        GdkCursor *cursor;

        cursor = gdk_cursor_new_for_display (display,
                                              widget == window->resize_sw ?
                                              GDK_BOTTOM_LEFT_CORNER :
                                              GDK_BOTTOM_RIGHT_CORNER);
        gdk_window_set_cursor (gdk_window, cursor);
        gdk_cursor_unref (cursor);
    } else {
        gdk_window_set_cursor (gdk_window, NULL);
    }
}

static gboolean
text_view_button_press (GtkWidget      *widget,
                        GdkEventButton *event,
                        EvAnnotationWindow *window)
{
    ev_annotation_window_grab_focus (window);

    return FALSE;
}

static void
ev_annotation_window_close (EvAnnotationWindow *window)
{
    gtk_widget_hide (GTK_WIDGET (window));
    g_signal_emit (window, signals[CLOSED], 0);
}

static void
ev_annotation_window_init (EvAnnotationWindow *window)
{
    GtkWidget *vbox, *hbox;
    GtkWidget *icon;
    GtkWidget *swindow;

    gtk_widget_set_can_focus (GTK_WIDGET (window), TRUE);

    vbox = gtk_vbox_new (FALSE, 0);

    /* Title bar */
    hbox = gtk_hbox_new (FALSE, 0);

    icon = gtk_image_new (); /* FIXME: use the annot icon */
    gtk_box_pack_start (GTK_BOX (hbox), icon, FALSE, FALSE, 0);
    gtk_widget_show (icon);

    window->title = gtk_label_new (NULL);
    gtk_box_pack_start (GTK_BOX (hbox), window->title, TRUE, TRUE, 0);
    gtk_widget_show (window->title);

    window->close_button = gtk_button_new ();
    gtk_button_set_relief (GTK_BUTTON (window->close_button), GTK_RELIEF_NONE);
    gtk_container_set_border_width (GTK_CONTAINER (window->close_button), 0);
    g_signal_connect_swapped (window->close_button, "clicked",
                              G_CALLBACK (ev_annotation_window_close),
                              window);

    icon = gtk_image_new_from_stock (EV_STOCK_CLOSE, EV_ICON_SIZE_ANNOT_WINDOW);
    gtk_container_add (GTK_CONTAINER (window->close_button), icon);
    gtk_widget_show (icon);

    gtk_box_pack_start (GTK_BOX (hbox), window->close_button, FALSE, FALSE, 0);
    gtk_widget_show (window->close_button);
}
```

```

gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, FALSE, 0);
gtk_widget_show (hbox);

/* Contents */
swindow = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (swindow),
                                GTK_POLICY_AUTOMATIC,
                                GTK_POLICY_AUTOMATIC);
window->text_view = gtk_text_view_new ();
gtk_text_view_set_wrap_mode (GTK_TEXT_VIEW (window->text_view), GTK_WRAP_WORD);
gtk_window_activate_default (window->text_view, FALSE);
g_signal_connect (window->text_view, "button_press_event",
                  G_CALLBACK (text_view_button_press),
                  window);
gtk_container_add (GTK_CONTAINER (swindow), window->text_view);
gtk_widget_show (window->text_view);

gtk_box_pack_start (GTK_BOX (vbox), swindow, TRUE, TRUE, 0);
gtk_widget_show (swindow);

/* Resize bar */
hbox = gtk_hbox_new (FALSE, 0);

window->resize_sw = gtk_event_box_new ();
gtk_widget_add_events (window->resize_sw, GDK_BUTTON_PRESS_MASK);
g_signal_connect_swapped (window->resize_sw, "button-press-event",
                          G_CALLBACK (ev_annotation_window_resize),
                          window);
g_signal_connect (window->resize_sw, "realize",
                  G_CALLBACK (ev_annotation_window_set_resize_cursor),
                  window);

icon = gtk_image_new_from_stock (EV_STOCK_RESIZE_SW, EV_ICON_SIZE_ANNOT_WINDOW);
gtk_container_add (GTK_CONTAINER (window->resize_sw), icon);
gtk_widget_show (icon);
gtk_box_pack_start (GTK_BOX (hbox), window->resize_sw, FALSE, FALSE, 0);
gtk_widget_show (window->resize_sw);

window->resize_se = gtk_event_box_new ();
gtk_widget_add_events (window->resize_se, GDK_BUTTON_PRESS_MASK);
g_signal_connect_swapped (window->resize_se, "button-press-event",
                          G_CALLBACK (ev_annotation_window_resize),
                          window);
g_signal_connect (window->resize_se, "realize",
                  G_CALLBACK (ev_annotation_window_set_resize_cursor),
                  window);

icon = gtk_image_new_from_stock (EV_STOCK_RESIZE_SE, EV_ICON_SIZE_ANNOT_WINDOW);
gtk_container_add (GTK_CONTAINER (window->resize_se), icon);
gtk_widget_show (icon);
gtk_box_pack_end (GTK_BOX (hbox), window->resize_se, FALSE, FALSE, 0);
gtk_widget_show (window->resize_se);

gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, FALSE, 0);
gtk_widget_show (hbox);

gtk_container_add (GTK_CONTAINER (window), vbox);
gtk_container_set_border_width (GTK_CONTAINER (window), 0);
gtk_widget_show (vbox);

gtk_widget_add_events (GTK_WIDGET (window),
                      GDK_BUTTON_PRESS_MASK |
                      GDK_KEY_PRESS_MASK);
gtk_widget_set_app_paintable (GTK_WIDGET (window), TRUE);

gtk_container_set_border_width (GTK_CONTAINER (window), 2);

gtk_window_set_accept_focus (GTK_WINDOW (window), TRUE);
gtk_window_set_decorated (GTK_WINDOW (window), FALSE);
gtk_window_set_skip_taskbar_hint (GTK_WINDOW (window), TRUE);
gtk_window_set_skip_pager_hint (GTK_WINDOW (window), TRUE);
gtk_window_set_resizable (GTK_WINDOW (window), TRUE);
}

static GObject *
ev_annotation_window_constructor (GType
                                guint
                                type,
                                n_construct_properties,
                                GObjectConstructParam *construct_params)

```

```

{
    GObject          *object;
    EvAnnotationWindow *window;
    EvAnnotation      *annot;
    gchar             *label;
    gdouble            opacity;
    EvRectangle        *rect;
    gdouble            scale;

    object = G_OBJECT_CLASS (ev_annotation_window_parent_class)->constructor (type,
n_construct_
properties,
rams);
    window = EV_ANNOTATION_WINDOW (object);
    annot = window->annotation;

    gtk_window_set_transient_for (GTK_WINDOW (window), window->parent);
    gtk_window_set_destroy_with_parent (GTK_WINDOW (window), FALSE);

    g_object_get (annot,
        "label", &label,
        "opacity", &opacity,
        "is_open", &window->is_open,
        "rectangle", &window->rect,
        NULL);
    rect = window->rect;

    /* Rectangle is at doc resolution (72.0) */
    scale = get_screen_dpi (window) / 72.0;
    gtk_window_resize (GTK_WINDOW (window),
        (gint)((rect->x2 - rect->x1) * scale),
        (gint)((rect->y2 - rect->y1) * scale));
    ev_annotation_window_set_color (window, &annot->color);
    gtk_widget_set_name (GTK_WIDGET (window), annot->name);
    gtk_window_set_title (GTK_WINDOW (window), label);
    gtk_label_set_text (GTK_LABEL (window->title), label);
    gtk_window_set_opacity (GTK_WINDOW (window), opacity);
    g_free (label);

    if (annot->contents) {
        GtkTextBuffer *buffer;

        buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (window->text_view));
        gtk_text_buffer_set_text (buffer, annot->contents, -1);
    }

    return object;
}

static gboolean
ev_annotation_window_button_press_event (GtkWidget      *widget,
GdkEventButton *event)
{
    EvAnnotationWindow *window = EV_ANNOTATION_WINDOW (widget);

    if (event->type == GDK_BUTTON_PRESS && event->button == 1) {
        window->in_move = TRUE;
        window->x = event->x;
        window->y = event->y;
        gtk_window_begin_move_drag (GTK_WINDOW (widget),
            event->button,
            event->x_root,
            event->y_root,
            event->time);

        return TRUE;
    }

    return FALSE;
}

static gboolean
ev_annotation_window_configure_event (GtkWidget      *widget,
GdkEventConfigure *event)
{
    EvAnnotationWindow *window = EV_ANNOTATION_WINDOW (widget);

    if (window->in_move &&

```

```

        (window->x != event->x || window->y != event->y)) {
            window->x = event->x;
            window->y = event->y;
        }

        return GTK_WIDGET_CLASS (ev_annotation_window_parent_class)->configure_event (widget,
event);
    }

static gboolean
ev_annotation_window_focus_in_event (GtkWidget      *widget,
                                     GdkEventFocus *event)
{
    EvAnnotationWindow *window = EV_ANNOTATION_WINDOW (widget);

    if (window->in_move) {
        window->orig_x = window->x;
        window->orig_y = window->y;
    }

    return FALSE;
}

static gboolean
ev_annotation_window_focus_out_event (GtkWidget      *widget,
                                      GdkEventFocus *event)
{
    EvAnnotationWindow *window = EV_ANNOTATION_WINDOW (widget);

    if (window->in_move &&
        (window->orig_x != window->x || window->orig_y != window->y)) {
        window->in_move = FALSE;
        g_signal_emit (window, signals[Moved], 0, window->x, window->y);
    }

    return FALSE;
}

static void
ev_annotation_window_class_init (EvAnnotationWindowClass *klass)
{
    GObjectClass *g_object_class = G_OBJECT_CLASS (klass);
    GtkWidgetClass *gtk_widget_class = GTK_WIDGET_CLASS (klass);

    g_object_class->constructor = ev_annotation_window_constructor;
    g_object_class->set_property = ev_annotation_window_set_property;
    g_object_class->dispose = ev_annotation_window_dispose;

    gtk_widget_class->button_press_event = ev_annotation_window_button_press_event;
    gtk_widget_class->configure_event = ev_annotation_window_configure_event;
    gtk_widget_class->focus_in_event = ev_annotation_window_focus_in_event;
    gtk_widget_class->focus_out_event = ev_annotation_window_focus_out_event;

    g_object_class_install_property (g_object_class,
PROP_ANNOTATION,
g_param_spec_object ("annotation",
"Annotation",
"The annotation associated to th
e window",
EV_TYPE_ANNOTATION_MARKUP,
G_PARAM_WRITABLE |
G_PARAM_CONSTRUCT_ONLY));

    g_object_class_install_property (g_object_class,
PROP_PARENT,
g_param_spec_object ("parent",
"Parent",
"The parent window",
GTK_TYPE_WINDOW,
G_PARAM_WRITABLE |
G_PARAM_CONSTRUCT_ONLY));

    signals[CLOSED] =
g_signal_new ("closed",
G_TYPE_FROM_CLASS (g_object_class),
G_SIGNAL_RUN_LAST | G_SIGNAL_ACTION,
G_STRUCT_OFFSET (EvAnnotationWindowClass, closed),
NULL, NULL,
g_cclosure_marshal_VOID__VOID,
G_TYPE_NONE, 0, G_TYPE_NONE);

```

```
signals[MOVED] =
    g_signal_new ("moved",
        G_TYPE_FROM_CLASS (g_object_class),
        G_SIGNAL_RUN_LAST | G_SIGNAL_ACTION,
        G_STRUCT_OFFSET (EvAnnotationWindowClass, moved),
        NULL, NULL,
        ev_view_marshal_VOID__INT_INT,
        G_TYPE_NONE, 2,
        G_TYPE_INT, G_TYPE_INT);
}

/* Public methods */
GtkWidget *
ev_annotation_window_new (EvAnnotation *annot,
                          GtkWidget *parent)
{
    GtkWidget *window;

    g_return_val_if_fail (EV_IS_ANNOTATION_MARKUP (annot), NULL);
    g_return_val_if_fail (GTK_IS_WINDOW (parent), NULL);

    window = g_object_new (EV_TYPE_ANNOTATION_WINDOW,
                          "annotation", annot,
                          "parent", parent,
                          NULL);

    return window;
}

EvAnnotation *
ev_annotation_window_get_annotation (EvAnnotationWindow *window)
{
    g_return_val_if_fail (EV_IS_ANNOTATION_WINDOW (window), NULL);

    return window->annotation;
}

void
ev_annotation_window_set_annotation (EvAnnotationWindow *window,
                                     EvAnnotation *annot)
{
    g_return_if_fail (EV_IS_ANNOTATION_WINDOW (window));
    g_return_if_fail (EV_IS_ANNOTATION (annot));

    if (annot == window->annotation)
        return;

    g_object_unref (window->annotation);
    window->annotation = g_object_ref (annot);
    ev_annotation_window_check_contents_modified (window);
    g_object_notify (G_OBJECT (window), "annotation");
}

gboolean
ev_annotation_window_is_open (EvAnnotationWindow *window)
{
    g_return_val_if_fail (EV_IS_ANNOTATION_WINDOW (window), FALSE);

    return window->is_open;
}

const EvRectangle *
ev_annotation_window_get_rectangle (EvAnnotationWindow *window)
{
    g_return_val_if_fail (EV_IS_ANNOTATION_WINDOW (window), NULL);

    return window->rect;
}

void
ev_annotation_window_set_rectangle (EvAnnotationWindow *window,
                                    EvRectangle *rect)
{
    g_return_if_fail (EV_IS_ANNOTATION_WINDOW (window));
    g_return_if_fail (rect != NULL);

    *window->rect = *rect;
}
```



```
void
ev_annotation_window_grab_focus (EvAnnotationWindow *window)
{
    g_return_if_fail (EV_IS_ANNOTATION_WINDOW (window));

    if (!gtk_widget_has_focus (window->text_view)) {
        gtk_widget_grab_focus (GTK_WIDGET (window));
        send_focus_change (window->text_view, TRUE);
    }
}

void
ev_annotation_window_ungrab_focus (EvAnnotationWindow *window)
{
    g_return_if_fail (EV_IS_ANNOTATION_WINDOW (window));

    if (gtk_widget_has_focus (window->text_view)) {
        send_focus_change (window->text_view, FALSE);
    }

    ev_annotation_window_check_contents_modified (window);
}
```