23. Scribble, A Simple Example Drawing Program

23.1 Overview

In this section, we will build a simple drawing program. In the process, we will examine how to handle mouse events, how to draw in a window, and how to do drawing better by using a backing pixmap. After creating the simple drawing program, we will extend it by adding support for XInput devices, such as drawing tablets. GTK provides support routines which makes getting extended information, such as pressure and tilt, from such devices quite easy.

23.2 Event Handling

The GTK signals we have already discussed are for high-level actions, such as a menu item being selected. However, sometimes it is useful to learn about lower-level occurrences, such as the mouse being moved, or a key being pressed. There are also GTK signals corresponding to these low-level *events*. The handlers for these signals have an extra parameter which is a pointer to a structure containing information about the event. For instance, motion event handlers are passed a pointer to a GdkEventMotion structure which looks (in part) like:

```
struct _GdkEventMotion
{
  GdkEventType type;
  GdkWindow *window;
  guint32 time;
  gdouble x;
  gdouble y;
  ...
  guint state;
  ...
};
```

`type` will be set to the event type, in this case `GDK_MOTION_NOTIFY`, window is the window in which the event occurred. `x` and `y` give the coordinates of the event. `state` specifies the modifier state when the event occurred (that is, it specifies which modifier keys and mouse buttons were pressed). It is the bitwise OR of some of the following:

```
GDK_SHIFT_MASK
GDK_LOCK_MASK
GDK_CONTROL_MASK
GDK_MOD1_MASK
GDK_MOD2_MASK
GDK_MOD3_MASK
GDK_MOD4_MASK
GDK_MOD5_MASK
GDK_BUTTON1_MASK
GDK_BUTTON2_MASK
GDK_BUTTON3_MASK
GDK_BUTTON4_MASK
GDK_BUTTON5_MASK
```

As for other signals, to determine what happens when an event occurs we call `gtk_signal_connect()`. But we also need let GTK know which events we want to be notified about. To do this, we call the function:

```
void gtk_widget_set_events (GtkWidget *widget,
                            gint      events);
```

The second field specifies the events we are interested in. It is the bitwise OR of constants that specify different types of events. For future reference the event types are:

```
GDK_EXPOSURE_MASK
GDK_POINTER_MOTION_MASK
GDK_POINTER_MOTION_HINT_MASK
GDK_BUTTON_MOTION_MASK
GDK_BUTTON1_MOTION_MASK
GDK_BUTTON2_MOTION_MASK
GDK_BUTTON3_MOTION_MASK
GDK_BUTTON_PRESS_MASK
GDK_BUTTON_RELEASE_MASK
GDK_KEY_PRESS_MASK
GDK_KEY_RELEASE_MASK
GDK_ENTER_NOTIFY_MASK
GDK_LEAVE_NOTIFY_MASK
GDK_FOCUS_CHANGE_MASK
GDK_STRUCTURE_MASK
GDK_PROPERTY_CHANGE_MASK
GDK_PROXIMITY_IN_MASK
GDK_PROXIMITY_OUT_MASK
```

There are a few subtle points that have to be observed when calling `gtk_widget_set_events()`. First, it must be called before the X window for a GTK widget is created. In practical terms, this means you should call it immediately after creating the widget. Second, the widget must have an associated X window. For efficiency, many widget types do not have their own window, but draw in their parent's window. These widgets are:

```
GtkAlignment
GtkArrow
GtkBin
GtkBox
GtkImage
GtkItem
GtkLabel
GtkPixmap
GtkScrolledWindow
GtkSeparator
GtkTable
GtkAspectFrame
GtkFrame
GtkVBox
GtkHBox
GtkVSeparator
GtkHSeparator
```

To capture events for these widgets, you need to use an EventBox widget. See the section on the EventBox widget for details.

For our drawing program, we want to know when the mouse button is pressed and when the mouse is moved, so we specify `GDK_POINTER_MOTION_MASK` and `GDK_BUTTON_PRESS_MASK`. We also want to know when we need to redraw our window, so we specify `GDK_EXPOSURE_MASK`. Although we want to be notified via a Configure event when our window size changes, we don't have to specify the corresponding `GDK_STRUCTURE_MASK` flag, because it is automatically specified for all windows.

It turns out, however, that there is a problem with just specifying `GDK_POINTER_MOTION_MASK`. This will cause the server to add a new motion event to the event queue every time the user moves the mouse. Imagine that it takes us 0.1 seconds to handle a motion event, but the X server queues a new motion event every 0.05 seconds. We will soon get way behind the users drawing. If the user draws for 5 seconds, it will take us another 5 seconds to catch up after they release the mouse button! What we would like is to only get one motion event for each event we process. The way to do this is to specify `GDK_POINTER_MOTION_HINT_MASK`.

When we specify `GDK_POINTER_MOTION_HINT_MASK`, the server sends us a motion event the first time the pointer moves after entering our window, or after a button press or release event. Subsequent motion events will be suppressed until we explicitly ask for the position of the pointer using the function:

```
GdkWindow*    gdk_window_get_pointer     (GdkWindow       *window,
                                          gint            *x,
                                          gint            *y,
                                          GdkModifierType *mask);
```

(There is another function, `gtk_widget_get_pointer()` which has a simpler interface, but turns out not to be very useful, since it only retrieves the position of the mouse, not whether the buttons are pressed.)

The code to set the events for our window then looks like:

```
  gtk_signal_connect (GTK_OBJECT (drawing_area), "expose_event",
                      (GtkSignalFunc) expose_event, NULL);
  gtk_signal_connect (GTK_OBJECT(drawing_area),"configure_event",
                      (GtkSignalFunc) configure_event, NULL);
  gtk_signal_connect (GTK_OBJECT (drawing_area), "motion_notify_event",
                      (GtkSignalFunc) motion_notify_event, NULL);
  gtk_signal_connect (GTK_OBJECT (drawing_area), "button_press_event",
                      (GtkSignalFunc) button_press_event, NULL);

  gtk_widget_set_events (drawing_area, GDK_EXPOSURE_MASK
                         | GDK_LEAVE_NOTIFY_MASK
                         | GDK_BUTTON_PRESS_MASK
                         | GDK_POINTER_MOTION_MASK
                         | GDK_POINTER_MOTION_HINT_MASK);
```

We'll save the "expose_event" and "configure_event" handlers for later. The "motion_notify_event" and "button_press_event" handlers are pretty simple:

```
static gint
button_press_event (GtkWidget *widget, GdkEventButton *event)
{
  if (event->button == 1 && pixmap != NULL)
      draw_brush (widget, event->x, event->y);

  return TRUE;
}

static gint
motion_notify_event (GtkWidget *widget, GdkEventMotion *event)
{
  int x, y;
  GdkModifierType state;

  if (event->is_hint)
    gdk_window_get_pointer (event->window, &x, &y, &state);
  else
    {
      x = event->x;
      y = event->y;
      state = event->state;
    }

  if (state & GDK_BUTTON1_MASK && pixmap != NULL)
    draw_brush (widget, x, y);

  return TRUE;
}
```

23.3 The DrawingArea Widget, And Drawing

We now turn to the process of drawing on the screen. The widget we use for this is the DrawingArea widget. A drawing area widget is essentially an X window and nothing more. It is a blank canvas in which we can draw whatever we like. A drawing area is created using the call:

```
GtkWidget* gtk_drawing_area_new        (void);
```

A default size for the widget can be specified by calling:

```
void       gtk_drawing_area_size       (GtkDrawingArea      *darea,
                                        gint                 width,
                                        gint                 height);
```

This default size can be overridden, as is true for all widgets, by calling `gtk_widget_set_usize()`, and that, in turn, can be overridden if the user manually resizes the the window containing the drawing area.

It should be noted that when we create a DrawingArea widget, we are *completely* responsible for drawing the contents. If our window is obscured then uncovered, we get an exposure event and must redraw what was previously hidden.

Having to remember everything that was drawn on the screen so we can properly redraw it can, to say the least, be a nuisance. In addition, it can be visually distracting if portions of the window are cleared, then redrawn step by step. The solution to this problem is to use an offscreen *backing pixmap*. Instead of drawing directly to the screen, we draw to an image stored in server memory but not displayed, then when the image changes or new portions of the image are displayed, we copy the relevant portions onto the screen.

To create an offscreen pixmap, we call the function:

```
GdkPixmap* gdk_pixmap_new              (GdkWindow  *window,
                                        gint        width,
                                        gint        height,
                                        gint        depth);
```

The `window` parameter specifies a GDK window that this pixmap takes some of its properties from. `width` and `height` specify the size of the pixmap. `depth` specifies the *color depth*, that is the number of bits per pixel, for the new window. If the depth is specified as `-1`, it will match the depth of `window`.

We create the pixmap in our "configure_event" handler. This event is generated whenever the window changes size, including when it is originally created.

```
/* Backing pixmap for drawing area */
static GdkPixmap *pixmap = NULL;

/* Create a new backing pixmap of the appropriate size */
static gint
configure_event (GtkWidget *widget, GdkEventConfigure *event)
{
  if (pixmap)
    gdk_pixmap_unref(pixmap);

  pixmap = gdk_pixmap_new(widget->window,
                          widget->allocation.width,
                          widget->allocation.height,
                          -1);
  gdk_draw_rectangle (pixmap,
                      widget->style->white_gc,
                      TRUE,
                      0, 0,
                      widget->allocation.width,
                      widget->allocation.height);

  return TRUE;
```

```
}
```

The call to `gdk_draw_rectangle()` clears the pixmap initially to white. We'll say more about that in a moment.

Our exposure event handler then simply copies the relevant portion of the pixmap onto the screen (we determine the area we need to redraw by using the event->area field of the exposure event):

```
/* Redraw the screen from the backing pixmap */
static gint
expose_event (GtkWidget *widget, GdkEventExpose *event)
{
  gdk_draw_pixmap(widget->window,
                  widget->style->fg_gc[GTK_WIDGET_STATE (widget)],
                  pixmap,
                  event->area.x, event->area.y,
                  event->area.x, event->area.y,
                  event->area.width, event->area.height);

  return FALSE;
}
```

We've now seen how to keep the screen up to date with our pixmap, but how do we actually draw interesting stuff on our pixmap? There are a large number of calls in GTK's GDK library for drawing on *drawables*. A drawable is simply something that can be drawn upon. It can be a window, a pixmap, or a bitmap (a black and white image). We've already seen two such calls above, `gdk_draw_rectangle()` and `gdk_draw_pixmap()`. The complete list is:

```
gdk_draw_line ()
gdk_draw_rectangle ()
gdk_draw_arc ()
gdk_draw_polygon ()
gdk_draw_string ()
gdk_draw_text ()
gdk_draw_pixmap ()
gdk_draw_bitmap ()
gdk_draw_image ()
gdk_draw_points ()
gdk_draw_segments ()
```

See the reference documentation or the header file `<gdk/gdk.h>` for further details on these functions. These functions all share the same first two arguments. The first argument is the drawable to draw upon, the second argument is a *graphics context* (GC).

A graphics context encapsulates information about things such as foreground and background color and line width. GDK has a full set of functions for creating and modifying graphics contexts, but to keep things simple we'll just use predefined graphics contexts. Each widget has an associated style. (Which can be modified in a gtkrc file, see the section GTK's rc file.) This, among other things, stores a number of graphics contexts. Some examples of accessing these graphics contexts are:

```
widget->style->white_gc
widget->style->black_gc
widget->style->fg_gc[GTK_STATE_NORMAL]
widget->style->bg_gc[GTK_WIDGET_STATE(widget)]
```

The fields `fg_gc`, `bg_gc`, `dark_gc`, and `light_gc` are indexed by a parameter of type `GtkStateType` which can take on the values:

```
GTK_STATE_NORMAL,
GTK_STATE_ACTIVE,
GTK_STATE_PRELIGHT,
```

```
GTK_STATE_SELECTED,
GTK_STATE_INSENSITIVE
```

For instance, for `GTK_STATE_SELECTED` the default foreground color is white and the default background color, dark blue.

Our function `draw_brush()`, which does the actual drawing on the screen, is then:

```
/* Draw a rectangle on the screen */
static void
draw_brush (GtkWidget *widget, gdouble x, gdouble y)
{
  GdkRectangle update_rect;

  update_rect.x = x - 5;
  update_rect.y = y - 5;
  update_rect.width = 10;
  update_rect.height = 10;
  gdk_draw_rectangle (pixmap,
                      widget->style->black_gc,
                      TRUE,
                      update_rect.x, update_rect.y,
                      update_rect.width, update_rect.height);
  gtk_widget_draw (widget, &update_rect);
}
```

After we draw the rectangle representing the brush onto the pixmap, we call the function:

```
void       gtk_widget_draw                (GtkWidget           *widget,
                                           GdkRectangle        *area);
```

which notifies X that the area given by the `area` parameter needs to be updated. X will eventually generate an expose event (possibly combining the areas passed in several calls to `gtk_widget_draw()`) which will cause our expose event handler to copy the relevant portions to the screen.

We have now covered the entire drawing program except for a few mundane details like creating the main window.

23.4 Adding XInput support

It is now possible to buy quite inexpensive input devices such as drawing tablets, which allow drawing with a much greater ease of artistic expression than does a mouse. The simplest way to use such devices is simply as a replacement for the mouse, but that misses out many of the advantages of these devices, such as:

- Pressure sensitivity
- Tilt reporting
- Sub-pixel positioning
- Multiple inputs (for example, a stylus with a point and eraser)

For information about the XInput extension, see the XInput-HOWTO.

If we examine the full definition of, for example, the GdkEventMotion structure, we see that it has fields to support extended device information.

```
struct _GdkEventMotion
{
  GdkEventType type;
  GdkWindow *window;
  guint32 time;
```

```
  gdouble x;
  gdouble y;
  gdouble pressure;
  gdouble xtilt;
  gdouble ytilt;
  guint state;
  gint16 is_hint;
  GdkInputSource source;
  guint32 deviceid;
};
```

`pressure` gives the pressure as a floating point number between 0 and 1. `xtilt` and `ytilt` can take on values between -1 and 1, corresponding to the degree of tilt in each direction. `source` and `deviceid` specify the device for which the event occurred in two different ways. `source` gives some simple information about the type of device. It can take the enumeration values:

```
GDK_SOURCE_MOUSE
GDK_SOURCE_PEN
GDK_SOURCE_ERASER
GDK_SOURCE_CURSOR
```

`deviceid` specifies a unique numeric ID for the device. This can be used to find out further information about the device using the `gdk_input_list_devices()` call (see below). The special value `GDK_CORE_POINTER` is used for the core pointer device. (Usually the mouse.)

Enabling extended device information

To let GTK know about our interest in the extended device information, we merely have to add a single line to our program:

```
gtk_widget_set_extension_events (drawing_area, GDK_EXTENSION_EVENTS_CURSOR);
```

By giving the value `GDK_EXTENSION_EVENTS_CURSOR` we say that we are interested in extension events, but only if we don't have to draw our own cursor. See the section Further Sophistications below for more information about drawing the cursor. We could also give the values `GDK_EXTENSION_EVENTS_ALL` if we were willing to draw our own cursor, or `GDK_EXTENSION_EVENTS_NONE` to revert back to the default condition.

This is not completely the end of the story however. By default, no extension devices are enabled. We need a mechanism to allow users to enable and configure their extension devices. GTK provides the InputDialog widget to automate this process. The following procedure manages an InputDialog widget. It creates the dialog if it isn't present, and raises it to the top otherwise.

```
void
input_dialog_destroy (GtkWidget *w, gpointer data)
{
  *((GtkWidget **)data) = NULL;
}

void
create_input_dialog ()
{
  static GtkWidget *inputd = NULL;

  if (!inputd)
    {
      inputd = gtk_input_dialog_new();

      gtk_signal_connect (GTK_OBJECT(inputd), "destroy",
                          (GtkSignalFunc)input_dialog_destroy, &inputd);
      gtk_signal_connect_object (GTK_OBJECT(GTK_INPUT_DIALOG(inputd)->close_button),
                                 "clicked",
```

```
                                    (GtkSignalFunc)gtk_widget_hide,
                                    GTK_OBJECT(inputd));
        gtk_widget_hide ( GTK_INPUT_DIALOG(inputd)->save_button);

        gtk_widget_show (inputd);
      }
    else
      {
        if (!GTK_WIDGET_MAPPED(inputd))
          gtk_widget_show(inputd);
        else
          gdk_window_raise(inputd->window);
      }
  }
```

(You might want to take note of the way we handle this dialog. By connecting to the "destroy" signal, we make sure that we don't keep a pointer to dialog around after it is destroyed - that could lead to a segfault.)

The InputDialog has two buttons "Close" and "Save", which by default have no actions assigned to them. In the above function we make "Close" hide the dialog, hide the "Save" button, since we don't implement saving of XInput options in this program.

Using extended device information

Once we've enabled the device, we can just use the extended device information in the extra fields of the event structures. In fact, it is always safe to use this information since these fields will have reasonable default values even when extended events are not enabled.

Once change we do have to make is to call `gdk_input_window_get_pointer()` instead of `gdk_window_get_pointer`. This is necessary because `gdk_window_get_pointer` doesn't return the extended device information.

```
void gdk_input_window_get_pointer( GdkWindow       *window,
                                   guint32         deviceid,
                                   gdouble         *x,
                                   gdouble         *y,
                                   gdouble         *pressure,
                                   gdouble         *xtilt,
                                   gdouble         *ytilt,
                                   GdkModifierType *mask);
```

When calling this function, we need to specify the device ID as well as the window. Usually, we'll get the device ID from the `deviceid` field of an event structure. Again, this function will return reasonable values when extension events are not enabled. (In this case, `event->deviceid` will have the value `GDK_CORE_POINTER`).

So the basic structure of our button-press and motion event handlers doesn't change much - we just need to add code to deal with the extended information.

```
static gint
button_press_event (GtkWidget *widget, GdkEventButton *event)
{
  print_button_press (event->deviceid);

  if (event->button == 1 && pixmap != NULL)
    draw_brush (widget, event->source, event->x, event->y, event->pressure);

  return TRUE;
}

static gint
```

```
motion_notify_event (GtkWidget *widget, GdkEventMotion *event)
{
  gdouble x, y;
  gdouble pressure;
  GdkModifierType state;

  if (event->is_hint)
    gdk_input_window_get_pointer (event->window, event->deviceid,
                                  &x, &y, &pressure, NULL, NULL, &state);
  else
    {
      x = event->x;
      y = event->y;
      pressure = event->pressure;
      state = event->state;
    }

  if (state & GDK_BUTTON1_MASK && pixmap != NULL)
    draw_brush (widget, event->source, x, y, pressure);

  return TRUE;
}
```

We also need to do something with the new information. Our new `draw_brush()` function draws with a different color for each `event->source` and changes the brush size depending on the pressure.

```
/* Draw a rectangle on the screen, size depending on pressure,
   and color on the type of device */
static void
draw_brush (GtkWidget *widget, GdkInputSource source,
            gdouble x, gdouble y, gdouble pressure)
{
  GdkGC *gc;
  GdkRectangle update_rect;

  switch (source)
    {
    case GDK_SOURCE_MOUSE:
      gc = widget->style->dark_gc[GTK_WIDGET_STATE (widget)];
      break;
    case GDK_SOURCE_PEN:
      gc = widget->style->black_gc;
      break;
    case GDK_SOURCE_ERASER:
      gc = widget->style->white_gc;
      break;
    default:
      gc = widget->style->light_gc[GTK_WIDGET_STATE (widget)];
    }

  update_rect.x = x - 10 * pressure;
  update_rect.y = y - 10 * pressure;
  update_rect.width = 20 * pressure;
  update_rect.height = 20 * pressure;
  gdk_draw_rectangle (pixmap, gc, TRUE,
                      update_rect.x, update_rect.y,
                      update_rect.width, update_rect.height);
  gtk_widget_draw (widget, &update_rect);
}
```

Finding out more about a device

As an example of how to find out more about a device, our program will print the name of the device that generates each button press. To find out the name of a device, we call the function:

```
GList *gdk_input_list_devices               (void);
```

which returns a GList (a linked list type from the GLib library) of GdkDeviceInfo structures. The GdkDeviceInfo structure is defined as:

```
struct _GdkDeviceInfo
{
  guint32 deviceid;
  gchar *name;
  GdkInputSource source;
  GdkInputMode mode;
  gint has_cursor;
  gint num_axes;
  GdkAxisUse *axes;
  gint num_keys;
  GdkDeviceKey *keys;
};
```

Most of these fields are configuration information that you can ignore unless you are implementing XInput configuration saving. The fieldwe are interested in here is `name` which is simply the name that X assigns to the device. The other field that isn't configuration information is `has_cursor`. If `has_cursor` is false, then we we need to draw our own cursor. But since we've specified `GDK_EXTENSION_EVENTS_CURSOR`, we don't have to worry about this.

Our `print_button_press()` function simply iterates through the returned list until it finds a match, then prints out the name of the device.

```
static void
print_button_press (guint32 deviceid)
{
  GList *tmp_list;

  /* gdk_input_list_devices returns an internal list, so we shouldn't
     free it afterwards */
  tmp_list = gdk_input_list_devices();

  while (tmp_list)
    {
      GdkDeviceInfo *info = (GdkDeviceInfo *)tmp_list->data;

      if (info->deviceid == deviceid)
        {
          printf("Button press on device '%s'\n", info->name);
          return;
        }

      tmp_list = tmp_list->next;
    }
}
```

That completes the changes to "XInputize" our program.

Further sophistications

Although our program now supports XInput quite well, it lacks some features we would want in a full-featured application. First, the user probably doesn't want to have to configure their device each time they run the program, so we should allow them to save the device configuration. This is done by iterating through the return of `gdk_input_list_devices()` and writing out the configuration to a file.

To restore the state next time the program is run, GDK provides functions to change device configuration:

```
gdk_input_set_extension_events()
gdk_input_set_source()
gdk_input_set_mode()
gdk_input_set_axes()
gdk_input_set_key()
```

(The list returned from `gdk_input_list_devices()` should not be modified directly.) An example of doing this can be found in the drawing program gsumi. (Available from http://www.msc.cornell.edu/~otaylor/gsumi/) Eventually, it would be nice to have a standard way of doing this for all applications. This probably belongs at a slightly higher level than GTK, perhaps in the GNOME library.

Another major omission that we have mentioned above is the lack of cursor drawing. Platforms other than XFree86 currently do not allow simultaneously using a device as both the core pointer and directly by an application. See the XInput-HOWTO for more information about this. This means that applications that want to support the widest audience need to draw their own cursor.

An application that draws its own cursor needs to do two things: determine if the current device needs a cursor drawn or not, and determine if the current device is in proximity. (If the current device is a drawing tablet, it's a nice touch to make the cursor disappear when the stylus is lifted from the tablet. When the device is touching the stylus, that is called "in proximity.") The first is done by searching the device list, as we did to find out the device name. The second is achieved by selecting "proximity_out" events. An example of drawing one's own cursor is found in the "testinput" program found in the GTK distribution.
