Drawing in GTK+

Background

In order to understand how to draw in GTK, you first have to understand something about how GTK draws widgets, because how GTK draws widgets has an important role in how you design your drawing application. An understanding of how GTK draws widgets is also required if you ever plan to build your own custom widgets.

Windows and Clipping

Most windowing systems are designed around the idea that an application's visual display lies within a rectangular region on the screen called its *window*. The windowing system, e.g. Gnome or KDE or Explorer, does not automatically save the graphical content of an application's windows; instead it asks the application itself to *repaint*[1] its windows whenever it is needed. For example, if a window that is stacked below other windows gets raised to the top, then a client program has to repaint the area that was previously obscured. When the windowing system asks a client program to redraw part of a window, it sends an *exposure event* to the program that contains that window. An exposure event is simply an event sent from the underlying windowing system to a widget to notify it that it must redraw itself.

In this context, a "window" means "a rectangular region with automatic clipping", not a top-level application window. Clipping is the act of removing portions of a window that do not need to be redrawn, or looked at the other way, it is determining which are the only regions of a window that must be redrawn. In Figure 1, window A is below window B, which is below window C. If the user does something that "brings A forward" on the screen, then the only region of window A that must be redrawn is the portion of A in the set $A \cap (B \cup C)$, which is shaded in the figure.

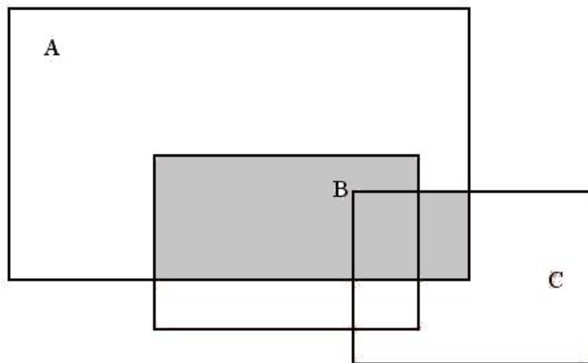

Figure 1: Window clipping.

Most windowing systems support nested windows, which are called *child windows*. A top-level window may contain many child windows, which may contain child windows in turn, and so on. For example, a top-level window will typically contain a window for the menu bar, one for the document area, one for each scrollbar,

[1] We will use the terms "draw" and "paint" interchangeably. Technically, the operation we are describing is painting, because it is the application of colors and brush strokes to a canvas. Drawing is just a special kind of painting. All GUI APIs refer to the process of visually rendering their windows as painting, or repainting, even if they describe the process occasionally as drawing. We will keep with this tradition of equating the two terms.

and one for the status bar. In addition, controls that receive user input, such as clickable buttons, typically have their own subwindows. It is possible for GTK+ to run on a windowing system with no notion of nested windows. This is not a problem, because GDK presents the illusion of being under such a system, and therefore, in GTK+, nested windows are always available.

The Drawing Cycle

Generally, the drawing cycle begins when GTK+ receives an exposure event from the underlying windowing system, such as when the user drags a window over another one. In this case the windowing system will tell the underlying window that it needs to repaint itself. The drawing cycle can also be initiated when a widget itself decides that it needs to update its display. For example, when the user types a character in a `GtkEntry` widget, the entry asks GTK+ to queue a redraw operation for itself. In other words, a widget can call a function that requests that the windowing system itself send it an event to redraw itself! This simplifies the program, as you will soon see.

A `GdkWindow` represents a window from the underlying windowing system on which GTK+ is running. On X11 it corresponds to a (X) Window; on Win32, it corresponds to a `HANDLE`. It is the windowing system that generates events for these windows. These events are passed to the GDK interface, which translates these native events into `GdkEvent` structures and sends them on to the GTK layer. In turn, the GTK widget layer finds the widget that corresponds to a particular `GdkWindow` and emits the corresponding event signals on that widget.

Widgets With and Without GdkWindows

If every single widget had its own `GdkWindow`, the drawing cycle would be trivial: the underlying windowing system would notify GDK about a window that needed redrawing, GDK would pass that exposure event for the specific `GdkWindow` to the GTK layer, and the GTK layer would emit the expose-event signal for that widget. The widget's expose event handler would then repaint the widget. Nothing else would have to be done; the windowing system would generate exposure events for each window that needed it, and then each corresponding widget would draw itself in turn.

However, in practice, there are reasons for which it is more convenient and efficient to allow widgets not to have a `GdkWindow` of their own, but to instead share the one from their parent widget. Every widget has a `GTK_NO_WINDOW` flag that indicates whether or not the widget has its own window. If a widget does not have its own window, it must inform GTK that it does not, when it is created, by setting this flag to `true`. (In GTK+-2.18 and later, the widget should call `gtk_widget_set_has_window()`, passing a `false` value in its second argument.) This is not something you would have to do as a programmer. It is the job of the widget's implementer to do this in the widget's constructor (its `init` method.) You, as a programmer, can test whether a particular widget has a `GdkWindow` of its own by inspecting the value of the `GTK_NO_WINDOW` flag or by calling the `gtk_widget_get_has_window()` method in GTK+-2.18 or later, which will return `true` if it has its own window[2]. Widgets that do not have their own GDK windows are called *no-window widgets* or `GTK_NO_WINDOW` widgets.

Why would you want a widget to be window-less? There are two primary reasons.

- Some widgets may want the parent's background to show through, even when they draw on parts of it. Such is the case, for instance, when a label is placed on a button with a themed texture. If each widget had its own window, and therefore its own background, labels would look bad because there would be a visible break with respect to their surroundings.
- Reducing the total number of GDK windows reduces the volume of traffic between GDK and GTK, so if a widget can be implemented without a window, it improves performance.

[2]You cannot just test whether the `window` member variable of the GdkWindow structure is NULL, because it may be sharing its parent's GdkWindow.

Hierarchical Drawing

With this understanding, we can direct our attention to describing the sequence of steps that take place when GTK receives the exposure event. The very first step is that GTK finds the widget that corresponds to the window that received the event. This widget cannot be a no-window widget, otherwise the widget wouldn't own the window that received the event. This widget begins by painting its background. Then, if it is a container widget, it tells each of its no-window children to paint themselves. This process is applied recursively for all the no-window descendants of the original widget.

This process does not get propagated to widgets that have windows of their own (those for which `GTK_NO_WINDOW` is `true`). This is because if any of those widgets require redrawing, the windowing system will have already sent exposure events to their corresponding GDK windows. Thus, there is no need to propagate the exposure to them on the GTK+ side.

An Example (from the GTK+ API Documentation)

The following example is from the online Gnome/GTK+ documentation, "The GTK+ Drawing Model," at http://library.gnome.org/devel/gtk/unstable/chap-drawing-model.html. It demonstrates how a top-level window would repaint itself when it has only no-window widget children.

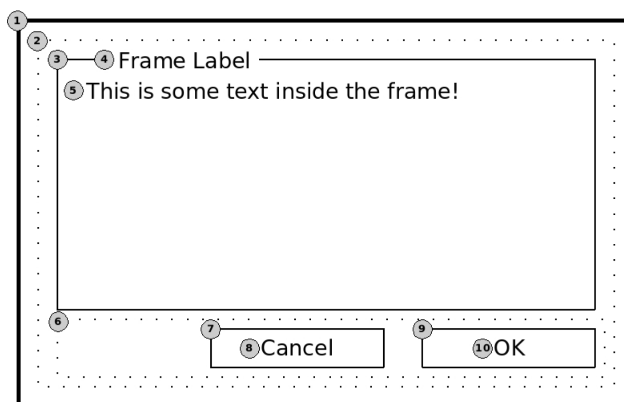

Figure 2: Hierarchical redrawing of a window

1. The outermost, thick rectangle is a top-level `GtkWindow`, which *is not* a `GTK_NO_WINDOW` widget — therefore, it does receive its exposure event as it comes from GDK. First the `GtkWindow` would paint its own background. Then, it would ask its only child to paint itself, which is the dotted-rectangle numbered 2.

2. The dotted rectangle represents a `GtkVBox`, which has been made the sole child of the `GtkWindow`. GtkBoxes, which are no-window widgets, are just layout containers that do not paint anything by themselves, so this `GtkVBox` would draw nothing, but rather ask its children to draw themselves. The children are numbered 3 and 6.

3. The thin rectangle is a `GtkFrame`, also a no-window widget, which has two children: a label for the frame, numbered 4, and another label inside, numbered 5. First the frame would draw its own beveled box, then ask the frame label and its internal child to draw themselves.

4. The frame label has no children, so it just draws its text: "Frame Label".

5. The internal label has no children, so it just draws its text: "This is some text inside the frame!".

6. The dotted rectangle represents a `GtkHBox`. Again, this does not draw anything by itself, but rather asks its children to draw themselves. The children are numbered 7 and 9.

7. The thin rectangle is a `GtkButton` with a single child, numbered 8. First the button would draw its beveled box, and then it would ask its child to draw itself.
8. This is a text label which has no children, so it just draws its own text: "Cancel".
9. Similar to number 7, this is a `GtkButton` with a single child, numbered 10. First the button would draw its beveled box, and then it would ask its child to draw itself.
10. Similar to number 8, this is a `GtkLabel` which has no children, so it just draws its own text: "OK".

The above steps illustrate how a widget is redrawn. If all of this drawing were to take place directly on the screen, there would be flickering as the various parts were redrawn. This is partly because many areas get redrawn repeatedly, such as backgrounds and decorative elements. Therefore, GTK implements a ***double buffering*** scheme at the GDK level.

Double buffering

In double buffering, there are two canvases: an off-screen canvas and an on-screen canvas. Drawing takes place on the off-screen canvas, which is then painted to the screen. The canvas that was on the screen then becomes the off-screen canvas for the next drawing. In GTK, this double buffering is opaque to the widgets; they normally don't know that they are drawing on an off-screen buffer; they just issue their normal drawing commands, and the buffer gets sent to the windowing system when all drawing operations are done.

Two basic functions in GDK form the core of the double-buffering mechanism:

- `gdk_window_begin_paint_region()`
- `gdk_window_end_paint()`.

The first function tells a `GdkWindow` to create a temporary off-screen buffer for drawing. All subsequent drawing operations to this window get automatically redirected to that buffer. The second function actually paints the buffer onto the on-screen window, and frees the buffer.

Automatic double buffering

It would be inconvenient for all widgets to have to call `gdk_window_begin_paint_region()` at the beginning of their expose handlers and `gdk_window_end_paint()` at the end. To make this easier, most GTK+ widgets have the `GTK_DOUBLE_BUFFERED` widget flag turned on by default. When GTK encounters such a widget, it automatically calls `gdk_window_begin_paint_region()` before emitting the expose-event signal for the widget, and then it calls `gdk_window_end_paint()` after the signal has been emitted. The terminology needs explanation here. The signal emission process ends when all handlers for the signal have been invoked. Remember that, in general there is a sequence of signal handler invocations when a signal is emitted. The callback that you supply is just one of them. Thus, after the expose event signal has been handled, the buffer is painted to the screen, whether or not your application did anything to it.

This is convenient for most widgets, so that they do not need to worry about creating their own temporary drawing buffers or about calling those functions. However, some widgets may prefer to disable this kind of automatic double buffering and do things on their own. To do this, the `GTK_DOUBLE_BUFFERED` flag must be turned off in the widget's constructor. Certain widgets, such as image viewers, would benefit from this, whereas widgets such as text viewers, which have their backgrounds drawn and redrawn repeatedly, would not.

Painting on Widgets

Usually, you will not want to modify the appearance of a widget by painting on top of it. The exception, of course is the `GtkDrawingArea` widget, whose intended purpose is to be drawn upon. However, if you do decide to modify the appearance of an event box or a top-level window, you will need to tell GTK that you intend to do so, because otherwise, GTK will overwrite your drawing when it responds to an exposure event. This is because the expose-event handlers for these widgets are run *after* any drawing that you might do. To circumvent this, you must turn on the `GTK_APP_PAINTABLE` flag in the widget. Itis helpful to understand the sequence of handler invocations, which is as follows:

1. Your own expose-event handler gets run. It paints something on the window or the event box.
2. The widget's default expose-event signal handler gets run. If `GTK_APP_PAINTABLE` is turned off (this is the default), your drawing will be overwritten. If that flag is turned on, the widget will not draw its default contents and preserve your drawing instead.
3. The expose-event handler for the parent class gets run. Since both `GtkWindow` and `GtkEventBox` are descendants of `GtkContainer`, their no-window children will be asked to draw themselves recursively, as described above.

In short, if you intend to draw on either the `GtkWindow` or the `GtkEventBox`, turn on the `GTK_APP_PAINTABLE` flag for them. You do not need to do anything to use the `GtkDrawingArea` widget, because it ignores this flag; its contents are never overwritten.

More About Events in GTK+

The preceding discussion about paintable widgets mentioned that your expose-event signal handler runs before the widget's default handler for this signal, and that after that, yet another handler, that of the parent class, would run. Until now, we did not give much thought to the idea that multiple signal handlers would run in response to an event, not did we care much about the order in which they ran. However, to get a grip on the drawing process, it is important to understand event propagation and the chain of signal handlers that run.

Remember that in a GTK+ program, all GDK events are passed to a `GtkWidget`, which emits a corresponding signal. You handle these events by connecting handlers to `GtkWidget` signals. The X server sends each X client a stream of events, and these events are sent and received in the order of their occurrence. GDK converts each XEvent it receives into a `GdkEvent`, then places events in a queue. GTK+ monitors GDK's event queue; for each event received, it decides which widget (if any) should receive the event. The `GtkWidget` base class defines signals for most event types (such as "button_press_event"); it also defines a generic "event" signal. The GTK+ main loop calls `gtk_widget_event()` to deliver an event to a widget; this function first emits the "event" signal, then emits a signal for the specific event type (if appropriate). Some events are handled in special ways.

In general, events go to the widget owning the `GdkWindow` in which the event occurred. There are exceptions, such as when widgets are made insensitive: events representing user interactions are not forwarded to insensitive widgets. Also, widgets with no associated `GdkWindow` do not originate events; X only sends events to widgets that have GDK windows. There is one exception: containers synthesize expose events for their windowless children.

The GTK+ main loop propagates certain events from child widgets to their parent containers. That is, for each event, a signal is emitted first from a child widget, then from its immediate parent, then from the parent's parent, up the tree of containment. Some events are not propagated.

Event propagation ends once a widget "handles" the event. This ensures that only one user-visible change results from any user action. Handlers for `GtkWidget`'s event signals must return a `gint` value. The last signal handler to run determines the return value of a signal emission. All GDK event signals are `GTK_RUN_LAST`

signals, meaning that the default handler always runs after the user-defined handlers, except for those connected with `g_signal_connect_after()`. Therefore, the return value of the event handling sequence comes from

- The last handler connected with `gtk_signal_connect_after()`, if any,
- Otherwise, the widget's default signal handler, if any,
- Otherwise, the last handler connected with `gtk_signal_connect()`, if any,
- Otherwise, the default return value is `FALSE`.

If the emission of an event signal returns `TRUE`, the GTK+ main loop will stop propagating the current event. If it returns `FALSE`, the main loop will propagate the event to the widget's parent. As noted above, each event results in two signal emissions: a generic "event" signal and a specific signal (such as "button_press_event" or "key_press_event"). If either emission returns `TRUE`, event propagation ends. The return value from the generic "event" signal has one additional effect: if `TRUE`, the second, more specific signal will not be emitted.

Drawables

All drawing must be done in a GDK *drawable*, represented by the GDK object known as a `GdkDrawable`. A drawable is a thing that supports drawing onto itself, and is one of either a `GdkWindow`, a `GdkPixmap`, or a `GdkBitmap` object. A `GdkDrawable` is a direct subclass of `GObject`, and inherits all of its methods. The drawing operations supported by drawables are extensive: you can draw points, line segments, arcs, shapes such as rectangles, polygons, and trapezoids, Pango layouts (of text), simple text strings, images, and glyphs. You can copy portions of one drawable into another, and copy `GdkPixbufs` into drawables., or export a region of a drawable into a `GdkPixbuf` or a `GdkImage`. All of this makes them very powerful objects.

All drawing operations act on a `GdkDrawable` and have a pointer to that `GdkDrawable` as their first parameter. Since drawing often involves deciding upon properties such as the color of the foreground (e.g., the line color), the background, the width of the line or brush, whether or not a closed object such as a rectangle should be filled, whether the drawing should have a transparent quality, and so on, GDK defines a structure called a *graphics context*, known as a `GdkGC` object. A `GdkGC` contains a number of drawing attributes such as foreground color, background color and line width, and is used to reduce the number of arguments needed for each drawing operation. Instead of passing all of these attributes as parameters to the drawing methods, a `GdkGC` can be configured and passed to the method instead. For example,

```
GdkGC  *temp_gc;
/* create a new graphics context based on the GDK window on which we will draw */
temp_gc = gdk_gc_new(my_gdk_drawable);

/* set foreground color to “color” in the GC assuming color is defined already */
gdk_gc_set_foreground( temp_gc, color  );
/* draw an arc of 360 degrees at given position */
gdk_draw_arc( my_gdk_drawable, temp_gc, TRUE, upper_left_x,
              upper_left_y, diameter, diameter,0, 360*64 );
```

sets the foreground color to be used in drawing a circle within a square whose upper-left corner is at position (upper_left_x, upper_left_y) in the specified drawable. The last two arguments specify the angle in $1/64^{th}$s of a degree of the start and end of the arc, thus $(0, 360 * 64)$ defines a circle.

Widgets

Although every widget other than no-window widgets has a GDK window, you cannot draw on any widget's GDK window without doing a lot of extra work. There are certain widgets that are designed specifically to be drawn upon, namely, the `GtkDrawingArea` and the `GtkLayout`. It is also possible to draw indirectly onto a `GtkImage` widget. Of these, the `GtkDrawingArea` is the one you will find most useful for general purpose drawing, and so we will focus on an overview of this widget.

The `GtkDrawingArea` widget is essentially an X window and nothing more. It is a blank canvas upon which you can draw whatever you like. Because it is intended for user drawing, GTK does not repaint it on expose-events; it is your complete responsibility to redraw the contents of the window when it has been obscured and then uncovered and emits an expose-event signal. In other words, if you draw some complicated scene in a `GtkDrawingArea`'s window, and it is hidden and then uncovered, your job is to redraw that same complicated scene all over again.

Since GTK is not redrawing your drawing, and an expose event is an indication that all or part of it must be redrawn, your application must be able to redraw everything that was drawn in the window before the window was obscured. This implies that your application has to remember what has been drawn. If all you are doing is drawing a fixed collection of shapes, this is easy enough. You can create a collection of data structures for storing all of the information needed to redraw exactly what was in the window before it was obscured. When the application draws in functions other than the expose handler, it records what it is drawing. When the expose handler runs, it draws everything that has been recorded. In this type of approach, which we will call the *direct drawing method*, the application draws directly into the `GtkDrawingArea`'s GDK window, which is automatically displayed on the screen when you are finished drawing.

Direct Drawing Method

The basic paradigm is illustrated by the following example. Suppose that `draw_star(widget, point)` draws a 5-pointed star of fixed size centered on the given point in the given drawable widget, and that a `PointArray` is defined by

```
typedef struct _point_array {
    GdkPoint  point[MAX_STARS];
    guint     length;
} PointArray;
```

Suppose also that the program should draw a star centered on the cursor when the user clicks the mouse button. The two functions in Listing 1 illustrate the idea. The button_press handler draws directly on the window and also records enough information so that the expose_event handler can redraw what was drawn in the button press handler.

Listing 1: Direct Drawing Method

```
static gboolean  on_expose  (GtkWidget       *area,
                             GdkEventExpose  *event,
                             PointArray      *centers )
{
    guint  i;
    for  (  i  =  0;  i  <  centers->length;  i++) {
        draw_star(GTK_WIDGET(area),   centers->point[i]   );
    }
    return TRUE;
}

static gboolean on_button_press  (GtkWidget      *area,
                                  GdkEventButton *event,
```

```
                                    PointArray      *star_centers )
{
    GdkPoint center = {event->x, event->y};

    if ( event->button == 1 ) {
        //draw a star whose center is at the cursor
        draw_star(GTK_WIDGET(area), center );

        // store the coordinates of the center for the expose handler
        star_centers->point[star_centers->length] = center;
        star_centers->length++;
    }
    return TRUE;
}
```

Off-Screen Pixmap Method

The direct drawing method is suitable for simple drawing applications. Having to remember everything that was drawn on the screen in order to redraw it can be a nuisance. In addition, it can be visually distracting if portions of the window are cleared, then redrawn step by step. Furthermore, it will not work if drawing is dynamic. For example, what if you want the application to implement "rubber-banding." In rubber-banding, a shape such as a line or rectangle is drawn and erased repeatedly while the mouse button is down and the mouse is being dragged until the user releases the mouse button, when the shape is added permanently to the window. The direct drawing method should not be used to implement rubber-banding because it would require an enormous amount of redrawing of the existing shapes and careful erasing of the rubber-banded shape as the mouse moved.

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. The image that we draw to is called a *pixmap*, short for pixel map. A pixmap is implemented in GDK by the `GdkPixmap` object. A `GdkPixmap` is a server-side resource. This means that it resides on the X-server's hosting machine. If the X-client application is running on a different machine, then there will be communication overhead in drawing to the pixmap. If there is extensive manipulation of the pixmap, the application may not behave well when used across a network.

A pixmap is created with `gdk_pixmap_new()`:

```
GdkPixmap*  gdk_pixmap_new (GdkDrawable *drawable,  // to determine depth of pixmap
                            gint width,             // width of pixmap in pixels
                            gint height,            // height of pixmap in pixels
                            gint depth);            // bits per pixel
```

To create a pixmap, you have to specify its depth. The depth of a pixmap is the number of bits per pixel. The safest way to create a pixmap is to let GDK decide on the depth, since the depth of the pixmap has to match the depth of the drawable in which it will be displayed eventually. Setting the depth parameter to -1 achieves this.

When you create the pixmap it is not initialized. Before drawing on it, you should paint a background color into it using the `gdk_draw_rectangle()` method, with fill set to `TRUE`, as in:

```
gdk_draw_rectangle (pixmap,
                    white_gc,
                    TRUE,
```

```
                              0, 0,
                              pixmap_width, pixmap_height );
```

where `white_gc` is a `GdkGC` in which the foregound color is white, and `pixmap_width` and `pixmap_height` are the width and height of the pixmap respectively. You can use all of the drawing primitives to draw on a pixmap, and it is not necessary to cast the pixmap to a different type, because the primitives all expect a `GdkDrawable` as their first argument, and a `GdkPixmap` is just a C *typedef* for a drawable.

When you are ready to copy the contents of a pixmap onto the on-screen drawable, for example, the `GtkDrawingArea`'s `GdkWindow` (e.g., `drawing_area->window`), you use the `gdk_draw_drawable()` method, which copies a source drawable into a target drawable:

```
void gdk_draw_drawable (GdkDrawable *target_drawable,
                        GdkGC *gc,
                        GdkDrawable *src,
                        gint xsrc,
                        gint ysrc,
                        gint xdest,
                        gint ydest,
                        gint width,
                        gint height);
```

This is probably self-explanatory. It copies the rectangle with upper-left corner at (`xsrc`,`ysrc`) in the drawable `src`, of width and height `width` and `height` respectively, into `target_drawable`, using the given graphics context for the drawing properties.

The off-screen pixmap method works as follows. In every function in which you would have drawn to the screen, you draw instead to the backing pixmap. Each time that you are finished drawing to the backing pixmap, you *invalidate* the rectangular region of the on-screen drawable's window that should contain the drawing you just did on the pixmap. Invalidating a window region causes an event to be placed in the event queue to redraw that region. In other words, it forces an expose event. Once the main loop becomes idle (after the current batch of events has been processed, roughly), the window will receive expose events for the union of all regions that have been invalidated. The function that invalidates the region is `gtk_widget_queue_draw_area()`:

```
void gtk_widget_queue_draw_area (GtkWidget *widget,
                                 gint x,
                                 gint y,
                                 gint width,
                                 gint height);
```

which invalidates the rectangular area of widget defined by `x`, `y`, `width` and `height`. In the expose handler for the application, you copy the backing pixmap to the on-screen window. Note that it is possible to just copy the backing pixmap to the on-screen window directly in all functions that draw into the pixmap, and not invalidate the drawn-upon region of the window. However, by invalidating the region and putting the copying in the expose handler, all copying occurs in one place, and time will be saved by avoiding multiple copying, because GDK consolidates the expose events into larger regions for a single expose event signal to be emitted by the widget, which will result one copy for multiple draws.

There is one other way in which this approach is different from the direct drawing method. If the user can resize the drawing area, then the backing pixmap will not match the drawing area in size after the resize operation. This is not a problem if the drawing area becomes smaller, but if it grows, then the pixmap will not be able to capture the drawing in the extended region of the drawing area. The solution is that the drawing area has to handle configure-events, which are generated when the user resizes the window. In the

handler for the configure-event, the application must check if the drawing area grew, and if so, create a new pixmap of the larger size, copy the old one into it, and then make the new one the backing pixmap. (It must also free the resources held by the old one, lest your program have a very large memory leak!)

In summary, there are three changes:

- you draw to a backing pixmap instead of the on-screen window and invalidate the drawn upon regions whenever you draw,
- your expose-event handler copies the pixmap to the on-screen window, and
- your configure-event handler resizes the pixmap and copies the old one to the new one.

The code in Listing 2 shows the various pieces of code that would replace the code in Listing 1.

Listing 2: Off-Screen Pixmap Method

```
gboolean on_configure_event( GtkWidget          *widget,
                             GdkEventConfigure  *event,
                             ApplicationState   *appState )
{
    gint current_width;
    gint current_height;
    GdkPixmap  *temp_pixmap;
    gint w = widget->allocation.width;
    gint h = widget->allocation.height;

    if (backing_pixmap == NULL ) {
        backing_pixmap = gdk_pixmap_new(widget->window, w, h, -1);
        gdk_draw_rectangle (backing_pixmap,
                            widget->style->white_gc,
                                TRUE,
                                0, 0,
                                w, h );
    }
    else {
        gdk_drawable_get_size(GDK_DRAWABLE(backing_pixmap),
                              &current_width, &current_height);
        if ( current_width < w || current_height < h ) {
            temp_pixmap = gdk_pixmap_new(widget->window, w, h, -1);

            // Fill it with white pixels
            gdk_draw_rectangle (temp_pixmap,
                                widget->style->white_gc,
                                  TRUE,
                                  0, 0, w, h );
            // Copy the old pixmap onto the new one using current foreground
            gdk_draw_drawable(temp_pixmap,
                              widget->style->fg_gc[GTK_WIDGET_STATE (widget)],
                              appState->backing_pixmap,
                              0, 0, 0, 0, -1, -1);

            // Release the reference to the old pixmap
            g_object_unref(appState->backing_pixmap);

            // and replace it with the new, properly-sized one
            appState->backing_pixmap = temp_pixmap;
        }
    }
    return TRUE;
```

```
}

static gboolean  on_expose (GtkWidget          *area,
                            GdkEventExpose     *event,
                            ApplicationState   *appState )
{
    gdk_draw_drawable(area->window,
        area->style->fg_gc[GTK_WIDGET_STATE(area)],
        appState->backing_pixmap,
        event->area.x,  event->area.y,
        event->area.x,  event->area.y,
        event->area.width,  event->area.height);

    return TRUE;
}

static gboolean  on_button_press  (GtkWidget       *area,
                                   GdkEventButton *event,
                                   ApplicationState   *appState )
{
    guint left, top, width, height;
    GdkPoint center = {event->x, event->y};

    if ( event->button == 1 ) {
        // Draw a star whose center is at the cursor
        // This fills left,top, width, and height with the rectangular
        // invalidate region
        draw_star(appState, center, &left, &top, &width, &height);

        // put an event in the event queue to force an expose event for the region
         gtk_widget_queue_draw_area(area, left, top, width, height);
    }
    return TRUE;
}
```

Rubber-Banding

Suppose that you want to extend the capabilities of your drawing applications to allow dynamically sized shapes. In particular, you want the user to be able to draw a star by specifying the bounding rectangle within which it should be displayed, so that the application can make a star as large as possible that fits the rectangle. Thus, as the user drags the mouse with the mouse button down, a dashed rectangle will be displayed, and when the use releases the mouse button, a star will appear within the rectangle. This is how many drawing tools let their users specify ellipses, circles, and other shapes. To simplify the drawing of the star, we would like to constrain the proportions of the rectangle so that it is always the same proportion. Since this task is incidental to the more important lesson, we will ignore this issue for now. The actual demo program does this, and you can look at its code. We will start with the simpler problem, namely, allowing the user to draw a rubber-banding rectangle and then drawing a star within it. Once we understand rubber-banding, we can return to the problem of constraining the rectangle.

The problem of rubber-banding is making the old rectangle disappear each time we move the mouse, and drawing a new one at the new mouse position. We will need to store the starting position of the mouse, i.e., the point at which the mouse button was pressed (but not released). Let us name this point, `start`. As the mouse moves, it defines a rectangle. Suppose the mouse is at position (x,y). Then it defines the rectangle whose diagonally opposite corners are `(start.x, start.y)` and `(x,y)`. (If `start.x = x` or `start.y = y`, then it is a line segment.) Each mouse movement while the button is held down defines a new rectangle.

Because every new position requires drwing a new rectangle, the drawing of the rectangle must take place in the motion-notify-event handler; this is the only handler that is invoked when the mouse moves.

Each time the mouse moves, it generates a motion-notify-event. The problem is that there will be way too many of these events to handle if we handle all of them, especially since we only want to know about those motion-notify-events that take place when the mouse button is down. We need a way to limit the number of messages that the application receives, to only those that take place when the button is held down. The solution is to use an *event mask*. We want to tell GTK to send us only those motion-notify-events that take place while the mouse button is down. More generally, we can tell GTK exactly which events we want by setting masks on the widget, using the `gtk_widget_set_events()` function:

```
gtk_widget_set_events (appState.drawing_area, GDK_EXPOSURE_MASK
                                              | GDK_BUTTON_PRESS_MASK
                                              | GDK_BUTTON_RELEASE_MASK
                                              | GDK_BUTTON_MOTION_MASK  );
```

This function must be called before the widget is realized. It sets which events the widget will receive. The exposure mask is needed to receive expose events, the button_motion mask, to receive motion-notify-events while any mouse button is down. The button press and release masks are needed so that we know when the mouse button is first pressed and when it is released.

To recap, we are going to draw the rectangle in the motion-event handler using the stored start position and the current event location as the rectangle's defining corners. This is not enough though, because we need to draw the star when the button is released, which is handled by the button-event handler. These will be two different callbacks. The button-event handler will also have to compute the bounding rectangle's coordinates and draw the star within it. Since there are at least two functions needing the start position, we put this information in the ApplicationState structure that we can pass to the handlers in the third parameter.

1. The problem of rubber-banding is solved by using a second, temporary pixmap. In the motion-notify-event handler, we do the following:
2. Get the current size of the drawing area widget.
3. Create a new temporary pixmap, say temp_pixmap, of the same size as the drawing area, and the same depth.
4. Copy the current contents of the backing pixmap into the temp_pixmap.
5. Draw a dashed rectangle into the temp_pixmap, one of whose corners is at (`event->x, event->y`) and the other, at (`start.x, start.y`). We cannot assume that the start point is the upper-left; it might be the upper-right, the lower-left, or the lower-right, depending where the mouse was moved.
6. Copy the temp_pixmap onto the on-screen drawable. To be efficient, we could restrict the copy to just the rectangular region containing the newly drawn rectangle. For simplicity, we just copy the whole temp_pixmap.
7. Free the temp_pixmap by calling `g_object_unref()` on it.

This completes the steps in the motion-notify-event handler. It leaves out the details of drawing the rectangle. You have to decide where the upper-left corner is by comparing the values of the start and end coordinates.

The button-event handler does a different task. It has to first check whether it was a button-press or a button release. If it is a button press event, it simply stores the position of the mouse in the application state[3]. If it is a button release event, then the star must be drawn into the backing pixmap. To determine the position of the star, the bounding rectangle is calculated the same way as in the motion-event handler,

[3] I use a separate MouseState structure usually, which contains the start and end positions, and whether the mouse button is up or down.

and the star is drawn within that rectangle. The star-drawing function must be modified to draw a star based on the position of the upper-left corner of the rectangle, and the width and height of the rectangle, i.e., its prototype would be

```
void draw_star (ApplicationState *appState,   // contains the backing pixmap
                gint left,                    // x-coordinate of upper-left corner
                gint top,                     // y-coordinate of upper-left corner
                gint width,                   // width of bounding rectangle
                gint height);                 // height of bounding rectangle
```

Once the star is drawn, a drawing event is put on the queue using `gtk_widget_queue_draw_area()`. The boundaries of the drawing event should match those of the bounding rectangle of the star.

This is the set of all changes. The various pieces of the program are contained in 3, 4, 5, and 6. The comments have been removed from these listings. The remainder of the program is in the demos directory on our host system.

Listing 3: Types and global constants for rubber-banding version

```
#define WIN_WIDTH    809
#define WIN_HEIGHT   500
#define LINE_WIDTH     1
#define STAR_PROPORTION   0.95105651629515

typedef struct _mouseState
{
    gint            start_x;
    gint            start_y;
    gint            end_x;
    gint            end_y;
    gboolean        isMouseDown;
} MouseState;

typedef struct _ApplicationState
{
    GtkWidget       *area;
    MouseState      mouseState;
    gint            line_width;
    GdkGC           *gcSolid;
    GdkGC           *gcDash;
    GdkPixmap       *backing_pixmap;
} ApplicationState;
```

The main application state is slim for this program. More serious applications would store more state. The motion-notify-event handler listing and the button-event handler listing include the logic to constraint the rectangle's proportions to match those of a bounding rectangle for a regular pentagonal star. It is worth studying carefully how this is accomplished. They also handle the cases when the mouse is above and/or to the left of the starting point.

Listing 4: Motion-notify-event handler for rubberbanding version

```
static gboolean on_motion_notify_event( GtkWidget *widget,
                                        GdkEventMotion *event,
                                        ApplicationState *appState )
{
    int x, y;
    GdkModifierType state;
    gint  w ;
    gint  h ;
```

```
    gint width, height;
    GdkPixmap    *temp_pixmap;

    state = event->state;
    x = event->x;
    y = event->y;

    if (state & GDK_BUTTON1_MASK && appState->mouseState.isMouseDown ) {
        gdk_drawable_get_size(appState->backing_pixmap, &w, &h );

        temp_pixmap = gdk_pixmap_new(widget->window, w, h, -1);
        gdk_draw_drawable(temp_pixmap,
                          widget->style->fg_gc[GTK_STATE_NORMAL],
                          appState->backing_pixmap, 0, 0, 0, 0, -1, -1);

        width  = abs(x - appState->mouseState.start_x);
        height = abs(y - appState->mouseState.start_y);

        if ( width >  STAR_PROPORTION * height  )
        { //wider than tall
            appState->mouseState.end_x = x;
            if ( y > appState->mouseState.start_y )
                appState->mouseState.end_y = appState->mouseState.start_y +
                        width * STAR_PROPORTION;
            else
                appState->mouseState.end_y = appState->mouseState.start_y -
                        width * STAR_PROPORTION;
        }
        else
        {
            appState->mouseState.end_y = y;
            if ( x > appState->mouseState.start_x )
                appState->mouseState.end_x = appState->mouseState.start_x +
                         height / STAR_PROPORTION;
            else
                appState->mouseState.end_x = appState->mouseState.start_x -
                         height / STAR_PROPORTION;
        }

        draw_box( temp_pixmap, appState->gcDash, &(appState->mouseState) );
        gdk_draw_drawable(widget->window, widget->style->fg_gc[GTK_STATE_NORMAL],
          temp_pixmap, 0, 0, 0, 0, -1, -1);

        g_object_unref(temp_pixmap);
    }
    return TRUE;
}
```

Listing 5: Button-event handler for rubberbanding version

```
gboolean on_button_event(             GtkWidget        *widget,
                                      GdkEventButton   *event,
                                      ApplicationState *appState )
{
    gint start_x, start_y;
    gint  end_x, end_y;
    gint width, height;
    gint x, y;
```

```
if ( event->type == GDK_BUTTON_PRESS ) {
    appState->mouseState.start_x      = event->x;
    appState->mouseState.start_y      = event->y;
    appState->mouseState.isMouseDown  = TRUE;
}
else if ( event->type == GDK_BUTTON_RELEASE ) {
    appState->mouseState.end_x       = event->x;
    appState->mouseState.end_y       = event->y;
    appState->mouseState.isMouseDown = FALSE;

    x = event->x;
    y = event->y;
    width  = abs(x - appState->mouseState.start_x);
    height = abs(y - appState->mouseState.start_y);

    if ( width > STAR_PROPORTION * height )
    {
        // The mouse has moved so that the box is too wide. We shift the
        // y-coordinate of endpoint to form the correctly shaped box
        appState->mouseState.end_x = x;
        if ( y > appState->mouseState.start_y )
            appState->mouseState.end_y = appState->mouseState.start_y +
                  width * STAR_PROPORTION;
        else
            appState->mouseState.end_y = appState->mouseState.start_y -
                  width * STAR_PROPORTION;
    }
    else
    {
        // The mouse has moved so that it is either just right, or the
        // box is too tall. We shift the x-coordinate of the endpoint
        // so that it is in the right proportion.
        appState->mouseState.end_y = y;
        if ( x > appState->mouseState.start_x )
            appState->mouseState.end_x = appState->mouseState.start_x +
                   height / STAR_PROPORTION;
        else
            appState->mouseState.end_x = appState->mouseState.start_x -
                   height / STAR_PROPORTION;
    }
    /*
       Now we have to make sure that we account for when the mouse moves
       to the left and/or upwards. We store the start and end point
       coordinates in local variables for convenience.
    */
    start_x  = appState->mouseState.start_x;
    end_x    = appState->mouseState.end_x;
    start_y  = appState->mouseState.start_y;
    end_y    = appState->mouseState.end_y;
    /*
       If the start point and end point are the exact same point, we bail
       out, since we do not want to draw a single point!
       Assuming they are not the same point, we test whether the start is
       to the left of the end and above it. If either test is false, we
       swap the positions of start and end.
    */
    if ( (start_x != end_x ) || (start_y != end_y ) )
    {
```

```
            if ( start_x > end_x) {
                appState->mouseState.start_x = end_x;
                appState->mouseState.end_x   = start_x;
            }
            if ( start_y > end_y) {
                appState->mouseState.start_y = end_y;
                appState->mouseState.end_y   = start_y;
            }

            // Recalculate the width and height based on any changes
            width  = abs(appState->mouseState.end_x -
                                      appState->mouseState.start_x);
            height = abs(appState->mouseState.end_y -
                                      appState->mouseState.start_y);
            // Call the draw_star function to draw the star with upper-left
            // corner at the new start point with given width and height
            draw_star ( appState,
                        appState->mouseState.start_x,
                        appState->mouseState.start_y,
                        width, height );
        }
        // Force an expose event for just the portion of the window containing
        // the newly drawn star, so that it gets copied from the backing pixmap
        // to the on-screen drawable.   It has to include the actual box, so
        // the width and height are each greater by the line width.
        gtk_widget_queue_draw_area(widget, appState->mouseState.start_x,
                        appState->mouseState.start_y,
                        width+appState->line_width,
                        height+appState->line_width);
    }
    else {
        // We ignore all other button events, such as 2button and 3button events
    }
    return TRUE;
}
```

The star drawing code is included here as well. Now that the center is not supplied as an argument, the center must be found. Geometric facts about pentagons are used to compute the side, radius, and apothem, needed to construct the star. Regular pentagons are amazing figures, bestowed with many interesting properties, and related to the golden ratio, ϕ, as well. This version does not use ϕ, although it could.

Listing 6: Star-drawing function for rubberbanding version

```
#define DOUBLE_SIN_36      1.17557050458495
#define APOTHEM            0.68819096023559
#define COS_72             0.30901699437494742410 2
#define COS_36             0.8090169943749474241
#define SIDE_TO_HEIGHT     0.6498393924658126523 1   // 2*sin(36) / (1 + cos(36))

static void draw_star              (ApplicationState *appState,
                                    gint left, gint top,
                                    gint width, gint height)
{
    GdkPoint vertex[5];
    GdkPoint center;
    GdkGC *drawing_gc;

    double side    = height * SIDE_TO_HEIGHT; // length of side of pentagon
    double radius  = side / DOUBLE_SIN_36;     // dist from center to any vertex
    double apothem = side * APOTHEM;     // dist from center to midpoint of side
```

```
    double rise      = radius * COS_72;    // dist from center to any diagonal
    center.x         = left + width/2;          // position of center
    center.y         = top + radius;

    vertex[0].x = center.x;
    vertex[0].y = center.y - radius;
    vertex[1].x = center.x - side/2;
    vertex[1].y = center.y + apothem;
    vertex[2].x = center.x + width/2;
    vertex[2].y = center.y - rise;
    vertex[3].x = center.x - width/2;
    vertex[3].y = center.y - rise;
    vertex[4].x = center.x + side/2;
    vertex[4].y = center.y + apothem;

    drawing_gc = appState->area->style->fg_gc[GTK_WIDGET_STATE(appState->area)];
    gdk_draw_polygon(appState->backing_pixmap, drawing_gc, TRUE, vertex, 5);
}
```