Changes that need to be done at the time of the switch

This section outlines porting tasks that you need to tackle when you get to the point that you actually build your application against GTK+ 3. Making it possible to prepare for these in GTK+ 2.24 would have been either impossible or impractical.

Replace size_request by get_preferred_width/height

The request-phase of the traditional GTK+ geometry management has been replaced by a more flexible height-for-width system, which is described in detail in the API documentation (see the section called “Height-for-width Geometry Management”). As a consequence, the ::size-request signal and vfunc has been removed from GtkWidgetClass. The replacement for `size_request()` can take several levels of sophistication:

- As a minimal replacement to keep current functionality, you can simply implement the `GtkWidgetClass.get_preferred_width()` and `GtkWidgetClass.get_preferred_height()` vfuncs by calling your existing `size_request()` function. So you go from

```
static void
my_widget_class_init (MyWidgetClass *class)
{
  GtkWidgetClass *widget_class = GTK_WIDGET_CLASS (class);

  /* ... */

  widget_class->size_request = my_widget_size_request;

  /* ... */
}
```

to something that looks more like this:

```
static void
my_widget_get_preferred_width (GtkWidget *widget,
                               gint      *minimal_width,
                               gint      *natural_width)
{
  GtkRequisition requisition;

  my_widget_size_request (widget, &requisition);

  *minimal_width = *natural_width = requisition.width;
}

static void
my_widget_get_preferred_height (GtkWidget *widget,
                                gint      *minimal_height,
                                gint      *natural_height)
{
  GtkRequisition requisition;

  my_widget_size_request (widget, &requisition);

  *minimal_height = *natural_height = requisition.height;
}

 /* ... */
```

```

static void
my_widget_class_init (MyWidgetClass *class)
{
  GtkWidgetClass *widget_class = GTK_WIDGET_CLASS (class);

  /* ... */

  widget_class->get_preferred_width = my_widget_get_preferred_width;
  widget_class->get_preferred_height = my_widget_get_preferred_height;

  /* ... */

}
```

Sometimes you can make things a little more streamlined by replacing your existing `size_request()` implementation by one that takes an orientation parameter:

```
static void
my_widget_get_preferred_size (GtkWidget      *widget,
                              GtkOrientation  orientation,
                               gint           *minimal_size,
                               gint           *natural_size)
{

  /* do things that are common for both orientations ... */

  if (orientation == GTK_ORIENTATION_HORIZONTAL)
    {
      /* do stuff that only applies to width... */

      *minimal_size = *natural_size = ...
    }
  else
   {
      /* do stuff that only applies to height... */

      *minimal_size = *natural_size = ...
   }
}

static void
my_widget_get_preferred_width (GtkWidget *widget,
                               gint      *minimal_width,
                               gint      *natural_width)
{
  my_widget_get_preferred_size (widget,
                                GTK_ORIENTATION_HORIZONTAL,
                                minimal_width,
                                natural_width);
}

static void
my_widget_get_preferred_height (GtkWidget *widget,
                                gint      *minimal_height,
                                gint      *natural_height)
{
  my_widget_get_preferred_size (widget,
                                GTK_ORIENTATION_VERTICAL,
                                minimal_height,
                                natural_height);
}

```

```
 /* ... */
```

- If your widget can cope with a small size, but would appreciate getting some more space (a common example would be that it contains ellipsizable labels), you can do that by making your `GtkWidgetClass.get_preferred_width()` / `GtkWidgetClass.get_preferred_height()` functions return a smaller value for *`minimal`* than for *`natural`*. For *`minimal`*, you probably want to return the same value that your `size_request()` function returned before (since `size_request()` was defined as returning the minimal size a widget can work with). A simple way to obtain good values for *`natural`*, in the case of containers, is to use `gtk_widget_get_preferred_width()` and `gtk_widget_get_preferred_height()` on the children of the container, as in the following example:

```
static void
gtk_fixed_get_preferred_height (GtkWidget *widget,
                                gint      *minimum,
                                gint      *natural)
{
  GtkFixed *fixed = GTK_FIXED (widget);
  GtkFixedPrivate *priv = fixed->priv;
  GtkFixedChild *child;
  GList *children;
  gint child_min, child_nat;

  *minimum = 0;
  *natural = 0;

  for (children = priv->children; children; children = children->next)
    {
      child = children->data;

      if (!gtk_widget_get_visible (child->widget))
        continue;

      gtk_widget_get_preferred_height (child->widget, &child_min, &child_nat);

      *minimum = MAX (*minimum, child->y + child_min);
      *natural = MAX (*natural, child->y + child_nat);
    }
}
```

- Note that the `GtkWidgetClass.get_preferred_width()` / `GtkWidgetClass.get_preferred_height()` functions only allow you to deal with one dimension at a time. If your `size_request()` handler is doing things that involve both width and height at the same time (e.g. limiting the aspect ratio), you will have to implement `GtkWidgetClass.get_preferred_height_for_width()` and `GtkWidgetClass.get_preferred_width_for_height()`.

- To make full use of the new capabilities of the height-for-width geometry management, you need to additionally implement the `GtkWidgetClass.get_preferred_height_for_width()` and `GtkWidgetClass.get_preferred_width_for_height()`. For details on these functions, see the section called “Height-for-width Geometry Management”.

Replace GdkRegion by cairo_region_t

Starting with version 1.10, cairo provides a region API that is equivalent to the GDK region API (which was itself copied from the X server). Therefore, the region API has been removed in GTK+ 3.

Porting your application to the cairo region API should be a straight find-and-replace task. Please refer to the following table:

Table 1.

GDK	cairo
GdkRegion	cairo_region_t
GdkRectangle	cairo_rectangle_int_t
`gdk_rectangle_intersect()`	this function is still there
`gdk_rectangle_union()`	this function is still there
`gdk_region_new()`	`cairo_region_create()`
`gdk_region_copy()`	`cairo_region_copy()`

gdk_region_destroy()	cairo_region_destroy()
gdk_region_rectangle()	cairo_region_create_rectangle()
gdk_region_get_clipbox()	cairo_region_get_extents()
gdk_region_get_rectangles()	cairo_region_num_rectangles() and cairo_region_get_rectangle()
gdk_region_empty()	cairo_region_is_empty()
gdk_region_equal()	cairo_region_equal()
gdk_region_point_in()	cairo_region_contains_point()
gdk_region_rect_in()	cairo_region_contains_rectangle()
gdk_region_offset()	cairo_region_translate()
gdk_region_union_with_rect()	cairo_region_union_rectangle()
gdk_region_intersect()	cairo_region_intersect()
gdk_region_union()	cairo_region_union()
gdk_region_subtract()	cairo_region_subtract()
gdk_region_xor()	cairo_region_xor()
gdk_region_shrink()	no replacement
gdk_region_polygon()	no replacement, use cairo paths instead

Replace GdkPixmap by cairo surfaces

The GdkPixmap object and related functions have been removed. In the cairo-centric world of GTK+ 3, cairo surfaces take over the role of pixmaps.

Example 114. Creating custom cursors

One place where pixmaps were commonly used is to create custom cursors:

```c
GdkCursor *cursor;
GdkPixmap *pixmap;
cairo_t *cr;
GdkColor fg = { 0, 0, 0, 0 };

pixmap = gdk_pixmap_new (NULL, 1, 1, 1);

cr = gdk_cairo_create (pixmap);
cairo_rectangle (cr, 0, 0, 1, 1);
cairo_fill (cr);
cairo_destroy (cr);

cursor = gdk_cursor_new_from_pixmap (pixmap, pixmap, &fg, &fg, 0, 0);

g_object_unref (pixmap);
```

The same can be achieved without pixmaps, by drawing onto an image surface:

```c
GdkCursor *cursor;
cairo_surface_t *s;
cairo_t *cr;
GdkPixbuf *pixbuf;

s = cairo_image_surface_create (CAIRO_FORMAT_A1, 3, 3);
cr = cairo_create (s);
cairo_arc (cr, 1.5, 1.5, 1.5, 0, 2 * M_PI);
cairo_fill (cr);
cairo_destroy (cr);

pixbuf = gdk_pixbuf_get_from_surface (NULL, s,
                                      0, 0, 0, 0,
                                      3, 3);

cairo_surface_destroy (s);

cursor = gdk_cursor_new_from_pixbuf (display, pixbuf, 0, 0);
cursor = gdk_cursor_new_from_pixbuf (display, pixbuf, 0, 0);

g_object_unref (pixbuf);
```

Replace GdkColormap by GdkVisual

For drawing with cairo, it is not necessary to allocate colors, and a GdkVisual provides enough information for cairo to handle colors in 'native' surfaces. Therefore, GdkColormap and related functions have been removed in GTK+ 3, and visuals are used instead. The colormap-handling functions of GtkWidget (`gtk_widget_set_colormap()`, etc) have been removed and `gtk_widget_set_visual()` has been added.

Example 115. Setting up a translucent window

You might have a screen-changed handler like the following to set up a translucent window with an alpha-channel:

```
static void
on_alpha_screen_changed (GtkWidget *widget,
                         GdkScreen *old_screen,
                         GtkWidget *label)
{
  GdkScreen *screen = gtk_widget_get_screen (widget);
  GdkColormap *colormap = gdk_screen_get_rgba_colormap (screen);

  if (colormap == NULL)
    colormap = gdk_screen_get_default_colormap (screen);

  gtk_widget_set_colormap (widget, colormap);
}
```

With visuals instead of colormaps, this will look as follows:

```
static void
on_alpha_screen_changed (GtkWindow *window,
                         GdkScreen *old_screen,
                         GtkWidget *label)
{
  GdkScreen *screen = gtk_widget_get_screen (GTK_WIDGET (window));
  GdkVisual *visual = gdk_screen_get_rgba_visual (screen);

  if (visual == NULL)
    visual = gdk_screen_get_system_visual (screen);

  gtk_widget_set_visual (window, visual);
}
```

GdkDrawable is gone

GdkDrawable has been removed in GTK+ 3, together with GdkPixmap and GdkImage. The only remaining drawable class is GdkWindow. For dealing with image data, you should use a cairo_surface_t or a GdkPixbuf.

GdkDrawable functions that are useful with windows have been replaced by corresponding GdkWindow functions:

Table 2. GdkDrawable to GdkWindow

GDK 2.x	GDK 3
`gdk_drawable_get_visual()`	`gdk_window_get_visual()`
`gdk_drawable_get_size()`	`gdk_window_get_width()` `gdk_window_get_height()`
`gdk_pixbuf_get_from_drawable()`	`gdk_pixbuf_get_from_window()`
`gdk_drawable_get_clip_region()`	`gdk_window_get_clip_region()`
`gdk_drawable_get_visible_region()`	`gdk_window_get_visible_region()`

Event filtering

If your application uses the low-level event filtering facilities in GDK, there are some changes you need to be aware of.

The special-purpose GdkEventClient events and the `gdk_add_client_message_filter()` and `gdk_display_add_client_message_filter()` functions have been removed. Receiving X11 ClientMessage events is still possible, using the general `gdk_window_add_filter()` API. A client message filter like

```
static GdkFilterReturn
message_filter (GdkXEvent *xevent, GdkEvent *event, gpointer data)
{
  XClientMessageEvent *evt = (XClientMessageEvent *)xevent;

  /* do something with evt ... */
}

 ...

message_type = gdk_atom_intern ("MANAGER", FALSE);
gdk_display_add_client_message_filter (display, message_type, message_filter, NULL);
```

then looks like this:

```
static GdkFilterReturn
event_filter (GdkXEvent *xevent, GdkEvent *event, gpointer data)
{
  XClientMessageEvent *evt;
  GdkAtom message_type;

  if (((XEvent *)xevent)->type != ClientMessage)
    return GDK_FILTER_CONTINUE;

  evt = (XClientMessageEvent *)xevent;
  message_type = XInternAtom (evt->display, "MANAGER", FALSE);

  if (evt->message_type != message_type)
    return GDK_FILTER_CONTINUE;

  /* do something with evt ... */
}

 ...

gdk_window_add_filter (NULL, message_filter, NULL);
```

One advantage of using an event filter is that you can actually remove the filter when you don't need it anymore, using `gdk_window_remove_filter()`.

The other difference to be aware of when working with event filters in GTK+ 3 is that GDK now uses XI2 by default when available. That means that your application does not receive core X11 key or button events. Instead, all input events are delivered as XIDeviceEvents. As a short-term workaround for this, you can force your application to not use XI2, with `gdk_disable_multidevice()`. In the long term, you probably want to rewrite your event filter to deal with XIDeviceEvents.

Backend-specific code

In GTK+ 2.x, GDK could only be compiled for one backend at a time, and the `GDK_WINDOWING_X11` or `GDK_WINDOWING_WIN32` macros could be used to find out which one you are dealing with:

```
#ifdef GDK_WINDOWING_X11
      if (timestamp != GDK_CURRENT_TIME)
```

```
        gdk_x11_window_set_user_time (gdk_window, timestamp);
#endif
#ifdef GDK_WINDOWING_WIN32
        /* ... win32 specific code ... */
#endif
```

In GTK+ 3, GDK can be built with multiple backends, and currently used backend has to be determined at runtime, typically using type-check macros on a GdkDisplay or GdkWindow. You still need to use the GDK_WINDOWING macros to only compile code referring to supported backends:

```
#ifdef GDK_WINDOWING_X11
      if (GDK_IS_X11_DISPLAY (display))
        {
          if (timestamp != GDK_CURRENT_TIME)
            gdk_x11_window_set_user_time (gdk_window, timestamp);
        }
      else
#endif
#ifdef GDK_WINDOWING_WIN32
      if (GDK_IS_WIN32_DISPLAY (display))
        {
          /* ... win32 specific code ... */
        }
      else
#endif
       {
         g_warning ("Unsupported GDK backend");
       }
```

If you used the pkg-config variable `target` to conditionally build part of your project depending on the GDK backend, for instance like this:

```
AM_CONDITIONAL(BUILD_X11, test `$PKG_CONFIG --variable=target gtk+-2.0` = "x11")
```

then you should now use the M4 macro provided by GTK+ itself:

```
GTK_CHECK_BACKEND([x11], [3.0.2], [have_x11=yes], [have_x11=no])
AM_CONDITIONAL(BUILD_x11, [test "x$have_x11" = "xyes"])
```

GtkPlug and GtkSocket

The GtkPlug and GtkSocket widgets are now X11-specific, and you have to include the `<gtk/gtkx.h>` header to use them. The previous section about proper handling of backend-specific code applies, if you care about other backends.

The GtkWidget::draw signal

The GtkWidget "expose-event" signal has been replaced by a new "draw" signal, which takes a cairo_t instead of an expose event. The cairo context is being set up so that the origin at (0, 0) coincides with the upper left corner of the widget, and is properly clipped.

Note

In other words, the cairo context of the draw signal is set up in 'widget coordinates', which is different from traditional expose event handlers, which always assume 'window coordinates'.

The widget is expected to draw itself with its allocated size, which is available via the new `gtk_widget_get_allocated_width()` and `gtk_widget_get_allocated_height()` functions. It is not necessary to check for `GTK_WIDGET_IS_DRAWABLE()`, since GTK+ already does this check before emitting the "draw" signal.

There are some special considerations for widgets with multiple windows. Expose events are window-specific, and widgets with multiple windows could expect to get an expose event for each window that needs to be redrawn. Therefore, multi-window expose event handlers typically look like this:

```
if (event->window == widget->window1)
  {
     /* ... draw window1 ... */
  }
else if (event->window == widget->window2)
  {
     /* ... draw window2 ... */
  }
...
```

In contrast, the "draw" signal handler may have to draw multiple windows in one call. GTK+ has a convenience function `gtk_cairo_should_draw_window()` that can be used to find out if a window needs to be drawn. With that, the example above would look like this (note that the 'else' is gone):

```
if (gtk_cairo_should_draw_window (cr, widget->window1)
  {
     /* ... draw window1 ... */
  }
if (gtk_cairo_should_draw_window (cr, widget->window2)
  {
     /* ... draw window2 ... */
  }
...
```

Another convenience function that can help when implementing ::draw for multi-window widgets is `gtk_cairo_transform_to_window()`, which transforms a cairo context from widget-relative coordinates to window-relative coordinates.

All GtkStyle drawing functions (`gtk_paint_box()`, etc) have been changed to take a cairo_t instead of a window and a clip area. ::draw implementations will usually just use the cairo context that has been passed in for this.

Example 116. A simple ::draw function

```
gboolean
gtk_arrow_draw (GtkWidget *widget,
                cairo_t   *cr)
{
  GtkStyleContext *context;
  gint x, y;
  gint width, height;
  gint extent;

  context = gtk_widget_get_style_context (widget);

  width = gtk_widget_get_allocated_width (widget);
  height = gtk_widget_get_allocated_height (widget);

  extent = MIN (width - 2 * PAD, height - 2 * PAD);
  x = PAD;
  y = PAD;

  gtk_render_arrow (context, rc, G_PI / 2, x, y, extent);
}
```

GtkProgressBar orientation

In GTK+ 2.x, GtkProgressBar and GtkCellRendererProgress were using the GtkProgressBarOrientation enumeration to specify their orientation and direction. In GTK+ 3, both the widget and the cell renderer implement GtkOrientable, and have an additional 'inverted' property to determine their direction. Therefore, a call to `gtk_progress_bar_set_orientation()` needs to be replaced by a pair of calls to `gtk_orientable_set_orientation()` and `gtk_progress_bar_set_inverted()`. The following values correspond:

Table 3.

GTK+ 2.x	GTK+ 3	
GtkProgressBarOrientation	**GtkOrientation**	**inverted**
GTK_PROGRESS_LEFT_TO_RIGHT	GTK_ORIENTATION_HORIZONTAL	FALSE
GTK_PROGRESS_RIGHT_TO_LEFT	GTK_ORIENTATION_HORIZONTAL	TRUE
GTK_PROGRESS_TOP_TO_BOTTOM	GTK_ORIENTATION_VERTICAL	FALSE
GTK_PROGRESS_BOTTOM_TO_TOP	GTK_ORIENTATION_VERTICAL	TRUE

Check your expand and fill flags

The behaviour of expanding widgets has changed slightly in GTK+ 3, compared to GTK+ 2.x. It is now 'inherited', i.e. a container that has an expanding child is considered expanding itself. This is often the desired behaviour. In places where you don't want this to happen, setting the container explicity as not expanding will stop the expand flag of the child from being inherited. See `gtk_widget_set_hexpand()` and `gtk_widget_set_vexpand()`.

If you experience sizing problems with widgets in ported code, carefully check the "expand" and "fill" flags of your boxes.

Scrolling changes

The default values for the "hscrollbar-policy" and "vscrollbar-policy" properties have been changed from 'never' to 'automatic'. If your application was relying on the default value, you will have explicitly set it explicitly.

The ::set-scroll-adjustments signal on GtkWidget has been replaced by the GtkScrollable interface which must be implemented by a widget that wants to be placed in a GtkScrolledWindow. Instead of emitting ::set-scroll-adjustments, the scrolled window simply sets the "hadjustment" and "vadjustment" properties.

GtkObject is gone

GtkObject has been removed in GTK+ 3. Its remaining functionality, the ::destroy signal, has been moved to GtkWidget. If you have non-widget classes that are directly derived from GtkObject, you have to make them derive from GInitiallyUnowned (or, if you don't need the floating functionality, GObject). If you have widgets that override the destroy class handler, you have to adjust your class_init function, since destroy is now a member of GtkWidgetClass:

```
GtkObjectClass *object_class = GTK_OBJECT_CLASS (class);

object_class->destroy = my_destroy;
```

becomes

```
GtkWidgetClass *widget_class = GTK_WIDGET_CLASS (class);

widget_class->destroy = my_destroy;
```

In the unlikely case that you have a non-widget class that is derived from GtkObject and makes use of the destroy functionality, you have to implement ::destroy yourself.

GtkEntryCompletion signal parameters

The "match-selected" and "cursor-on-match" signals were erroneously given the internal filter model instead of the users model. This oversight has been fixed in GTK+ 3; if you have handlers for these signals, they will likely need slight adjustments.

Resize grips

The resize grip functionality has been moved from GtkStatusbar to GtkWindow. Any window can now have resize grips, regardless whether it has a statusbar or not. The functions `gtk_statusbar_set_has_resize_grip()` and `gtk_statusbar_get_has_resize_grip()` have disappeared, and instead there are now `gtk_window_set_has_resize_grip()` and `gtk_window_get_has_resize_grip()`.

Prevent mixed linkage

Linking against GTK+ 2.x and GTK+ 3 in the same process is problematic and can lead to hard-to-diagnose crashes. The `gtk_init()` function in both GTK+ 2.22 and in GTK+ 3 tries to detect this situation and abort with a diagnostic message, but this check is not 100% reliable (e.g. if the problematic linking happens only in loadable modules).

Direct linking of your application against both versions of GTK+ is easy to avoid; the problem gets harder when your application is using libraries that are themselves linked against some version of GTK+. In that case, you have to verify that you are using a version of the library that is linked against GTK+ 3 library that is linked against GTK+ 3.

If you are using packages provided by a distributor, it is likely that parallel installable versions of the library exist for GTK+ 2.x and GTK+ 3, e.g for vte, check for vte3; for webkitgtk look for webkitgtk3, and so on.

Install GTK+ modules in the right place

Some software packages install loadable GTK+ modules such as theme engines, gdk-pixbuf loaders or input methods. Since GTK+ 3 is parallel-installable with GTK+ 2.x, the two GTK+ versions have separate locations for their loadable modules. The location for GTK+ 2.x is `libdir/gtk-2.0` (and its subdirectories), for GTK+ 3 the location is `libdir/gtk-3.0` (and its subdirectories).

For some kinds of modules, namely input methods and pixbuf loaders, GTK+ keeps a cache file with extra information about the modules. For GTK+ 2.x, these cache files are located in `sysconfdir/gtk-2.0`. For GTK+ 3, they have been moved to `libdir/gtk-3.0/3.0.0/`. The commands that create these cache files have been renamed with a -3 suffix to make them parallel-installable.

Note that GTK+ modules often link against libgtk, libgdk-pixbuf, etc. If that is the case for your module, you have to be careful to link the GTK+ 2.x version of your module against the 2.x version of the libraries, and the GTK+ 3 version against hte 3.x versions. Loading a module linked against libgtk 2.x into an application using GTK+ 3 will lead to unhappiness and must be avoided.

Theming changes

In GTK+ 3.0, GtkStyleContext was added to replace GtkStyle and the theming infrastructure available in 2.x. GtkStyleContext is an object similar in spirit to GtkStyle, as it contains theming information, although in a more complete and tokenized fashion. There are two aspects to switching to GtkStyleContext: porting themes and theme engines, and porting applications, libraries and widgets.

Migrating themes

From GTK+ 3.0 on, theme engines must implement GtkThemingEngine and be installed in `$libdir/gtk+-3.0/$GTK_VERSION/theming-engines`, and the files containing style information must be written in the CSS-like format that is understood by GtkCssProvider. For a theme named "Clearlooks", the CSS file parsed by default is `$datadir/themes/Clearlooks/gtk-3.0/gtk.css`, with possible variants such as the dark theme being named `gtk-dark.css` in the same directory.

If your theme RC file was providing values for GtkSettings, you can install a `settings.ini` keyfile along with the `gtk.css` to provide theme-specific defaults for settings.

Key themes have been converted to CSS syntax too. See the GtkCssProvider documentation information about the syntax. GTK+ looks for key themes in the file `$datadir/themes/theme/gtk-3.0/gtk-keys.css`, where *theme* is the current key theme name.

Migrating theme engines

Migrating a GtkStyle based engine to a GtkThemingEngine based one should be straightforward for most of the vfuncs. Besides a cleanup in the available paint methods and a simplification in the passed arguments (in favor of GtkStyleContext containing all the information), the available render methods resemble those of GtkStyle quite evidently. Notable differences include:

1. All variations of `gtk_paint_box()`, `gtk_paint_flat_box()`, `gtk_paint_shadow()`, `gtk_paint_box_gap()` and `gtk_paint_shadow_gap()` are replaced by `gtk_render_background()`, `gtk_render_frame()` and `gtk_render_frame_gap()`. The first function renders frameless backgrounds and the last two render frames in various forms.
2. `gtk_paint_resize_grip()` has been subsumed by `gtk_render_handle()` with a GTK_STYLE_CLASS_GRIP class set in the style context.
3. `gtk_paint_spinner()` disappears in favor of `gtk_render_activity()` with a GTK_STYLE_CLASS_SPINNER class set in the style context.

The list of available render methods is:

`gtk_render_background()`: Renders a widget/area background.
`gtk_render_frame()`: Renders a frame border around the given rectangle. Usually the detail of the border depends on the theme information, plus the current widget state.
`gtk_render_frame_gap()`: Renders a frame border with a gap on one side.
`gtk_render_layout()`: Renders a PangoLayout.
`gtk_render_handle()`: Renders all kind of handles and resize grips, depending on the style class.
`gtk_render_check()`: Render checkboxes.
`gtk_render_option()`: Render radiobuttons.
`gtk_render_arrow()`: Renders an arrow pointing to a direction.
`gtk_render_expander()`: Renders an expander indicator, such as in GtkExpander.
`gtk_render_focus()`: Renders the indication that a widget has the keyboard focus.
`gtk_render_line()`: Renders a line from one coordinate to another.
`gtk_render_slider()`: Renders a slider, such as in GtkScale.
`gtk_render_extension()`: Renders an extension that protrudes from a UI element, such as a notebook tab.
`gtk_render_activity()`: Renders an area displaying activity, be it a progressbar or a spinner.
`gtk_render_icon_pixbuf()`: Renders an icon into a GdkPixbuf.

One of the main differences to GtkStyle-based engines is that the rendered widget is totally isolated from the theme engine, all style information is meant to be retrieved from the GtkThemingEngine API, or from the GtkWidgetPath obtained from `gtk_theming_engine_get_path()`, which fully represents the rendered widget's hierarchy from a styling point of view.

The detail string available in GtkStyle-based engines has been replaced by widget regions and style classes. Regions are a way for complex widgets to associate different styles with different areas, such as even and odd rows in a treeview. Style classes allow sharing of style information between widgets, regardless of their type. Regions and style classes can be used in style sheets to associate styles, and them engines can also access them. There are several predefined classes and regions such as `GTK_STYLE_CLASS_BUTTON` or `GTK_STYLE_REGION_TAB` in `gtkstylecontext.h`, although custom widgets may define their own, which themes may attempt to handle.

Extending the CSS parser

In GtkStyle-based engines, GtkRCStyle provided ways to extend the gtkrc parser with engine-specific extensions. This has been replaced by `gtk_theming_engine_register_property()`, which lets a theme engine register new properties with an arbitrary type. While there is built-in support for most basic types, it is possible to use a custom parser for the property.

The installed properties depend on the "name" property, so they should be added in the `constructed()` vfunc. For example, if an engine with the name "Clearlooks" installs a "focus-color" property with the type GdkRGBA, the property `-Clearlooks-focus-color` will be registered and accepted in CSS like this:

```
GtkEntry {
  -Clearlooks-focus-color: rgba(255, 0, 0, 1.0);
}
```

Widget style properties also follow a similar syntax, with the widget type name used as a prefix. For example, the "focus-line-width" style property can be modified in CSS as `-GtkWidget-focus-line-width`.

Using the CSS file format

The syntax of RC and CSS files formats is obviously different. The CSS-like syntax will hopefully be much more familiar to many people, lowering the barrier for custom theming.

Instead of going through the syntax differences one-by-one, we will present a more or less comprehensive example and discuss how it can be translated into CSS:

Example 117. Sample RC code

```
style "default" {
        xthickness = 1
        ythickness = 1

        GtkButton::child-displacement-x = 1
        GtkButton::child-displacement-y = 1
        GtkCheckButton::indicator-size = 14

        bg[NORMAL]        = @bg_color
        bg[PRELIGHT]      = shade (1.02, @bg_color)
        bg[SELECTED]      = @selected_bg_color
        bg[INSENSITIVE]   = @bg_color
        bg[ACTIVE]        = shade (0.9, @bg_color)

        fg[NORMAL]        = @fg_color
        fg[PRELIGHT]      = @fg_color
        fg[SELECTED]      = @selected_fg_color
        fg[INSENSITIVE]   = darker (@bg_color)
        fg[ACTIVE]        = @fg_color

        text[NORMAL]      = @text_color
        text[PRELIGHT]    = @text_color
        text[SELECTED]    = @selected_fg_color
        text[INSENSITIVE] = darker (@bg_color)
        text[ACTIVE]      = @selected_fg_color

        base[NORMAL]      = @base_color
        base[PRELIGHT]    = shade (0.95, @bg_color)
        base[SELECTED]    = @selected_bg_color
        base[INSENSITIVE] = @bg_color
        base[ACTIVE]      = shade (0.9, @selected_bg_color)

        engine "clearlooks" {
                colorize_scrollbar = TRUE
                style = CLASSIC
        }
}

style "tooltips" {
        xthickness = 4
        ythickness = 4

        bg[NORMAL]        = @tooltip_bg_color
        fg[NORMAL]        = @tooltip_fg_color
}

style "button" {
        xthickness = 3
        ythickness = 3

        bg[NORMAL]        = shade (1.04, @bg_color)
        bg[PRELIGHT]      = shade (1.06, @bg_color)
        bg[ACTIVE]        = shade (0.85, @bg_color)
}

style "entry" {
        xthickness = 3
        ythickness = 3

        bg[SELECTED] = mix (0.4, @selected_bg_color, @base_color)
        fg[SELECTED] = @text_color

        engine "clearlooks" {
                focus_color = shade (0.65, @selected_bg_color)
        }
}

style "other" {
        bg[NORMAL] = #fff;
}

class "GtkWidget" style "default"
class "GtkEntry" style "entry"
widget_class "*<GtkButton>" style "button"
widget "gtk-tooltip*" style "tooltips"
widget_class "window-name.*.GtkButton" style "other"
```

would roughly translate to this CSS:

Example 118. CSS translation

```
* {
  padding: 1;
  -GtkButton-child-displacement-x: 1;
  -GtkButton-child-displacement-y: 1;
  -GtkCheckButton-indicator-size: 14;

  background-color: @bg_color;
  color: @fg_color;

  -Clearlooks-colorize-scrollbar: true;
  -Clearlooks-style: classic;
}

*:hover {
  background-color: shade (@bg_color, 1.02);
}

*:selected {
  background-color: @selected_bg_color;
  color: @selected_fg_color;
}

*:insensitive {
  color: shade (@bg_color, 0.7);
}

*:active {
  background-color: shade (@bg_color, 0.9);
}

.tooltip {
  padding: 4;

  background-color: @tooltip_bg_color;
  color: @tooltip_fg_color;
}

.button {
  padding: 3;
  background-color: shade (@bg_color, 1.04);
}

.button:hover {
  background-color: shade (@bg_color, 1.06);
}

.button:active {
  background-color: shade (@bg_color, 0.85);
}

.entry {
  padding: 3;

  background-color: @base_color;
  color: @text_color;
}

.entry:selected {
  background-color: mix (@selected_bg_color, @base_color, 0.4);
  -Clearlooks-focus-color: shade (0.65, @selected_bg_color)
}

/* The latter selector is an specification of the first,
   since any widget may use the same classes or names */
#window-name .button,
GtkWindow#window-name GtkButton.button {
  background-color: #fff;
}
```

One notable difference is the reduction from fg/bg/text/base colors to only foreground/background, in exchange the widget is able to render its various elements with different CSS classes, which can be themed independently.

In the same vein, the light, dark and mid color variants that were available in GtkStyle should be replaced by a combination of symbolic colors and custom CSS, where necessary. text_aa should really not be used anywhere, anyway, and the white and black colors that were available in GtkStyle can just be replaced by literal GdkRGBA structs.

Access to colors has also changed a bit. With GtkStyle, the common way to access colors is:

```
GdkColor *color1;
GdkColor color2;

color1 = &style->bg[GTK_STATE_PRELIGHT];
gtk_style_lookup_color (style, "focus_color", &color2);
```

With GtkStyleContext, you generally use GdkRGBA instead of GdkColor and the code looks like this:

```
GdkRGBA *color1;
GdkRGBA  color2;

gtk_style_context_get (context, GTK_STATE_FLAG_PRELIGHT,
                       "background-color", &color1,
                       NULL);
gtk_style_context_lookup_color (context, "focus_color", &color2);

...

gdk_rgba_free (color1);
```

Note that the memory handling here is different: `gtk_style_context_get()` expects the address of a GdkRGBA* and returns a newly allocated struct, `gtk_style_context_lookup_color()` expects the address of an existing struct, and fills it.

It is worth mentioning that the new file format does not support custom keybindings nor stock icon mappings as the RC format did.

A checklist for widgets

When porting your widgets to use GtkStyleContext, this checklist might be useful.

1. Replace "style-set" handlers with "style-updated" handlers.
2. Try to identify the role of what you're rendering with any number of classes. This will replace the detail string. There is a predefined set of CSS classes which you can reuse where appropriate. Doing so will give you theming 'for free', whereas custom classes will require extra work in the theme. Note that complex widgets are likely to need different styles when rendering different parts, and style classes are one way to achieve this. This could result in code like the following (simplified) examples:

   Example 119. Setting a permanent CSS class

   ```
   static void
   gtk_button_init (GtkButton *button)
   {
     GtkStyleContext *context;

     ...

     context = gtk_widget_get_style_context (GTK_WIDGET (button));

     /* Set the "button" class */
     gtk_style_context_add_class (context, GTK_STYLE_CLASS_BUTTON);
   }
   ```

   Or

Example 120. Using dynamic CSS classes for different elements

```
static gboolean
gtk_spin_button_draw (GtkSpinButton *spin,
                      cairo_t       *cr)
{
  GtkStyleContext *context;

  ...

  context = gtk_widget_get_style_context (GTK_WIDGET (spin));

  gtk_style_context_save (context);
  gtk_style_context_add_class (context, GTK_STYLE_CLASS_ENTRY);

  /* Call to entry draw impl with "entry" class */
  parent_class->draw (spin, cr);

  gtk_style_context_restore (context);
  gtk_style_context_save (context);

  /* Render up/down buttons with the "button" class */
  gtk_style_context_add_class (context, GTK_STYLE_CLASS_BUTTON);
  draw_up_button (spin, cr);
  draw_down_button (spin, cr);

  gtk_style_context_restore (context);

  ...
}
```

Note that GtkStyleContext only provides fg/bg colors, so text/base is done through distinctive theming of the different classes. For example, an entry would usually be black on white while a button would usually be black on light grey.

3. Replace all `gtk_paint_*()` calls with corresponding `gtk_render_*()` calls.

   The most distinctive changes are the use of GtkStateFlags to represent the widget state and the lack of GtkShadowType. Note that widget state is now passed implicitly via the context, so to render in a certain state, you have to temporarily set the state on the context, as in the following example:

   Example 121. Rendering with a specific state

```
gtk_style_context_save (context);
gtk_style_context_set_state (context, GTK_STATE_FLAG_ACTIVE);
gtk_render_check (context, cr, x, y, width, height);
gtk_style_context_restore (context);
```

   For `gtk_render_check()` and `gtk_render_option()`, the *shadow_type* parameter is replaced by the GTK_STATE_FLAG_ACTIVE and GTK_STATE_FLAG_INCONSISTENT state flags. For things such as pressed/unpressed button states, GTK_STATE_FLAG_ACTIVE is used, and the CSS may style normal/active states differently to render outset/inset borders, respectively.

4. The various `gtk_widget_modify_*()` functions to override colors or fonts for individual widgets have been replaced by similar `gtk_widget_override_*()` functions.
5. It is no longer necessary to call `gtk_widget_style_attach()`, `gtk_style_attach()`, `gtk_style_detach()` or `gtk_widget_ensure_style()`.
6. Replace all uses of xthickness/ythickness. GtkStyleContext uses the CSS box model, and there are border-width/padding/margin properties to replace the different applications of X and Y thickness. Note that all of this is merely a guideline. Widgets may choose to follow it or not.

Parsing of custom resources

As a consequence of the RC format going away, calling `gtk_rc_parse()` or `gtk_rc_parse_string()` won't have any effect on a widgets appearance. The way to replace these calls is using a custom GtkStyleProvider, either for an individual widget through `gtk_style_context_add_provider()` or for all widgets on a screen through `gtk_style_context_add_provider_for_screen()`. Typically, the provider will be a GtkCssProvider, which parse CSS information from a file or from a string.

Example 122. Using a custom GtkStyleProvider

```
GtkStyleContext *context;
GtkCssProvider *provider;

context = gtk_widget_get_style_context (widget);
provider = gtk_css_provider_new ();
gtk_css_provider_load_from_data (GTK_CSS_PROVIDER (provider),
                                 ".frame1 {\n"
                                 "   border-image: url('gradient1.png') 10 10 10 10 stretch;\n"
                                 "}\n", -1, NULL);
gtk_style_context_add_provider (context,
                                GTK_STYLE_PROVIDER (provider),
                                GTK_STYLE_PROVIDER_PRIORITY_APPLICATION);
g_object_unref (provider);
```

Notice that you can also get style information from custom resources by implementing the GtkStyleProvider interface yourself. This is an advanced feature that should be rarely used.

Bonus points

There are some features in GtkStyleContext that were not available in GtkStyle, or were made available over time for certain widgets through extending the detail string in obscure ways. There is a lot more information available when rendering UI elements, and it is accessible in more uniform, less hacky ways. By going through this list you'll ensure your widget is a good citizen in a fully themable user interface.

1. If your widget renders a series of similar elements, such as tabs in a GtkNotebook or rows/column in a GtkTreeView, consider adding regions through `gtk_style_context_add_region()`. These regions can be referenced in CSS and the :nth-child pseudo-class may be used to match the elements depending on the flags passed.

   Example 123. Theming widget regions

   ```
   GtkNotebook tab {
     background-color: #f3329d;
   }

   GtkTreeView row::nth-child (even) {
     background-color: #dddddd;
   }
   ```

2. If your container renders child widgets within different regions, make it implement GtkContainer::`get_path_for_child()`. This function lets containers assign a special GtkWidgetPath to child widgets depending on their role/region. This is necessary to extend the concept above throughout the widget hierarchy.

   For example, a GtkNotebook modifies the tab labels' GtkWidgetPath so the "tab" region is added. This makes it possible to theme tab labels through:

   Example 124. Theming a widget within a parent container region

   ```
   GtkNotebook tab GtkLabel {
     font: Sans 8;
   }
   ```

3. If you intend several visual elements to look interconnected, make sure you specify rendered elements' connection areas with `gtk_style_context_set_junction_sides()`. It is of course up to the theme to make use of this information or not.
4. GtkStyleContext supports implicit animations on state changes for the most simple case out-of-the-box: widgets with a single animatable area, whose state is changed with `gtk_widget_set_state_flags()` or `gtk_widget_unset_state_flags()`. These functions trigger animated transitions for the affected state flags. Examples of widgets for which this kind of animation may be sufficient are GtkButton or GtkEntry.

   If your widget consists of more than a simple area, and these areas may be rendered with different states, make sure to mark the rendered areas with `gtk_style_context_push_animatable_region()` and `gtk_style_context_pop_animatable_region()`.

   `gtk_style_context_notify_state_change()` may be used to trigger a transition for a given state. The region ID will determine the animatable region that is affected by this transition.
