10

Graphics

Graphics in GNOME inhabit the full gamut of libraries, from native X to the GNOME libraries, following a steady progression from raw and hard to use, to abstract and easy to use. Xlib offers the lowest level through its drawing primitives. GDK in turn wraps Xlib, taking away a lot of the hassle behind color and pixel management, and adding some powerful rendering functions with the GdkRGB module. GNOME adds libart to the mix, a sophisticated image manipulation suite that GNOME uses to manipulate its own graphics extensions, such as the gdk-pixbuf image-loading library and the GNOME Canvas drawing surface (see Chapter 11). These layers are for the most part cumulative, so even the highest, most abstracted layers, like the GNOME Canvas, can still use libart, GDK, and Xlib directly and indirectly. However, it is good practice to restrict your API calls to adjacent layers when possible.

10.1 Graphics in the X Window System

Dealing with graphics at the level of the X Window System can be a lot of work. You have to do most of the color and buffer management yourself. The low-level nature of the Xlib API forces you to know intimate details of how video cards react to the various color depths. You should never have to touch Xlib directly in a GNOME or GTK+ application; in this section we'll concentrate on the basic concepts of color and buffer management, leaving the C code for later sections.

10.1.1 Frame Buffers

A frame buffer is the lowest level of the X11 graphics system. It sits closest to the video hardware, holding the raw data that the video card renders onto the

monitor screen. The video card's current color resolution determines the exact format of the data in the frame buffer. Each visible pixel on the screen takes up a varying number of consecutive bits in the frame buffer.

For example, a monochrome display with no variance in brightness would require only one on-off bit for each pixel, resulting in 8 screen pixels per byte of memory in the frame buffer. Conversely, a high-color display showing 16 million colors would require 24 bits to express the full range of color, leading to 3 bytes for *each* pixel in the frame buffer. Thus given a screen size of 640 × 480, the monochrome display would require a frame buffer of 38,400 bytes ([640 × 480 × 1 bit] ÷ 8), while the 16-million-color display would require 921,600 bytes ([640 × 480 × 24 bits] ÷ 8), or close to a full megabyte.

Another difference between monochrome and color monitors is the number of electron guns the monitor fires at the CRT screen. A monochrome monitor has a single gun, which means that the video card has to give it only a single value: the intensity of the single color. A color monitor is a little more complex; it uses three electron guns simultaneously—one for each of the colors red, green, and blue. Every color that shows up on the monitor screen is a combination of these three basic components. Each pixel on a color monitor requires three discrete values, one for each electron gun.

To get the most out of each user's graphics hardware, the X Window System must be able to handle these different situations gracefully. As we will see in the sections that follow, X defines several different display modes to cover these variations. Some display modes, or visuals (see Section 10.1.3), pass their color values directly to the frame buffer (and thus the electron guns); other visuals use indirect color lookup tables to reduce the number of simultaneous colors the frame buffer must juggle. In the next section we'll see how these color tables work.

10.1.2 Color Maps

One way that X can account for limited hardware is to restrict the number of colors an application is allowed to display. Even though the video card might be physically able to display all 16 million colors, it might not have enough on-board video RAM to store all the pixels for a given screen size. An older video card with only 1MB of VRAM would not have a frame buffer big enough to display an 800 × 600 screen in 16 million colors ([800 × 600 × 24 bits] ÷ 8 = 1,440,000 bytes, or 1.4MB).

However, this does not mean that you can't run 800 × 600 resolution with that video card. With a little clever color juggling, the video card can narrow the total number of colors it displays simultaneously, without restricting the total range of colors from which it can choose. The video card can choose, for

example, 256 important colors out of the 16 million possible, and store them in a lookup table, or **color map.** Rather than storing the entire 24-bit values of each pixel directly in the frame buffer, it can instead store the 24-bit color values in the lookup table, and put only the 8-bit lookup indexes into the frame buffer (see Figure 10.1). Thus in this case the video card has access to the full range of 16 million colors—only 256 of which it can display at any given time—but it suffers a frame buffer size of only 480,000 bytes ([800 × 600 × 8 bits] ÷ 8).

Notice that with a color lookup table, the video card can change colors indirectly, by altering the values stored in the table. Changing an entry in the lookup table changes the color for all pixels in the frame buffer that point to that entry. This trick can be exploited for some interesting color-cycling effects, but in most cases it leads to annoying color corruption and flashing.

While all of this may seem rather low-level and immaterial to a book on application programming, the effects of these hardware properties filter up through the various layers of abstractions to the application level. You, as an application developer, need to know about color maps and visuals if you want to implement certain features with graphics buffers. Even if you don't use color maps directly, you need to know a thing or two about color handling.

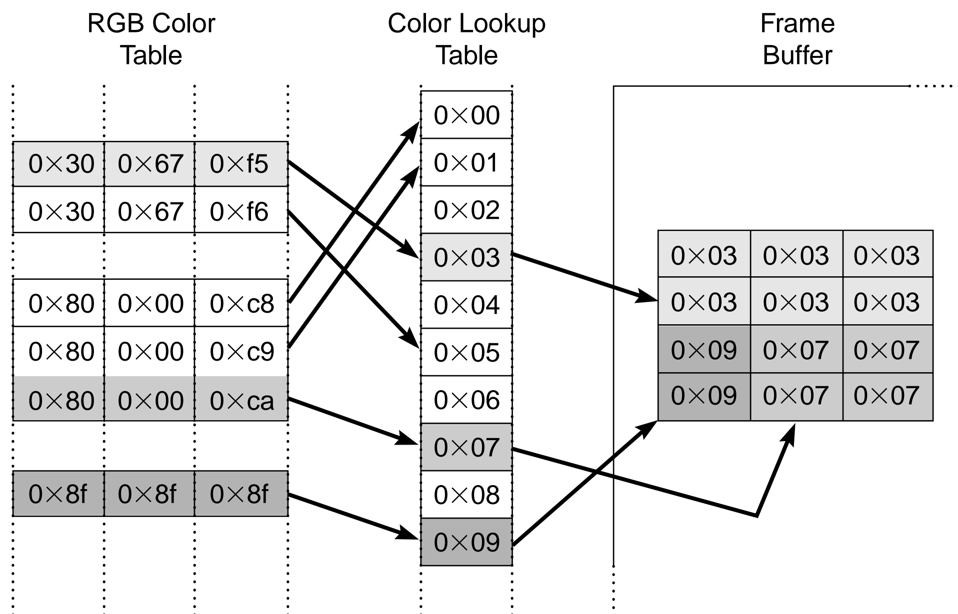


Figure 10.1 Color Map Example

10.1.3 Visuals

To simplify and categorize the various color-mapping situations, X11 divides them into six major groups, called **visuals.** Each visual represents a different style for handling color maps. A visual is a conceptual abstraction that allows us to describe complex color-mapping arrangements in clear, simple language. In theory, each top-level window can have its own visual, which implies that each window can have a different color depth. In practice, however, only a few higher-end X servers support per-window color depths; the other X servers limit all visuals on a display to the same color depth.

The six common visuals in X are StaticGray, GrayScale, StaticColor, PseudoColor, TrueColor, and DirectColor. These visuals have many interesting relationships with each other (see Figure 10.2). Their two primary characteristics are color depth and read/write access. Visuals can use a monochrome color map (StaticGray and GrayScale), or use an RGB color map as we discussed in the previous section (StaticColor and PseudoColor), or write colors directly to the frame buffer for full-color displays (TrueColor and DirectColor).

The other main distinction among visuals is whether or not their color map entries can be changed by the application. A static color map is read-only. Since the color map will never change, it can be shared by multiple applications without the risk of one application secretly changing the color palette used by another application. This sharing also cuts down on the amount of

Read/Write Access	Color Depth: Monochrome	RGB Color Map	Full Color
Read-Only	StaticGray	StaticColor	TrueColor
Read/Write	GrayScale	PseudoColor	DirectColor

Figure 10.2 Relationships among X Visuals

memory required to store color maps. Conversely, applications can change dynamic (read/write) color maps at any time. If an application uses a dynamic color map, it will typically allocate its own private color map so that other applications can't reappropriate its colors. Half of the six visuals—one of each color depth—have static color maps (StaticGray, StaticColor, and TrueColor), and the other half have dynamic color maps (GrayScale, PseudoColor, and DirectColor).

10.1.4 Drawables

Now that we've established a method for handling colors, we need to figure out how to show these colors on the screen. As we learned in Section 1.3.5, the X server keeps track of various server-side resources in a remote cache so that the client won't have to keep sending the same data over the wire. Thus the client can send a graphical image to the server one time, and the server will maintain its copy until the client explicitly requests its destruction. In the meantime, whenever the window is moved or covered up and reexposed, the X server can use its local resource copy to refresh the screen display rather than requesting a new copy of the image from the client each time.

The X Window System supports two types of these **drawables:** windows and pixmaps. A drawable is simply a server-side resource you can draw on with the various Xlib drawing commands. A **window** drawable is visible and makes up most of the on-screen real estate used by applications. The application can show or hide windows at any time. When you draw to a window, the effects of that operation show up immediately on the screen.

The other type of drawable is the **pixmap.** The pixmap is very similar to the window. You can invoke the same drawing commands on a pixmap that you can on a window. Unlike windows, however, pixmaps are never visible. Your drawing commands change the contents of the server-side pixmap resource, but those contents are not rendered to the screen. To see them, you must copy them to a window.

One of the most common uses for pixmaps is **double buffering,** in which the application uses a pixmap as an intermediary drawing buffer to smooth out the drawing process (see Figure 10.3). A complex graphic might require a long series of drawing commands to complete. Each time the window is reexposed, the client normally has to reexecute the same drawing commands. If the client is displaying over a slow connection, or if the rendering process takes too long, the graphic flickers as the display updates in real time. On the other hand, if the client renders to an off-screen pixmap, it doesn't matter how long the process takes. The client can update the pixmap at its leisure, whenever the graphic changes. When the X server needs to refresh the on-screen image, the

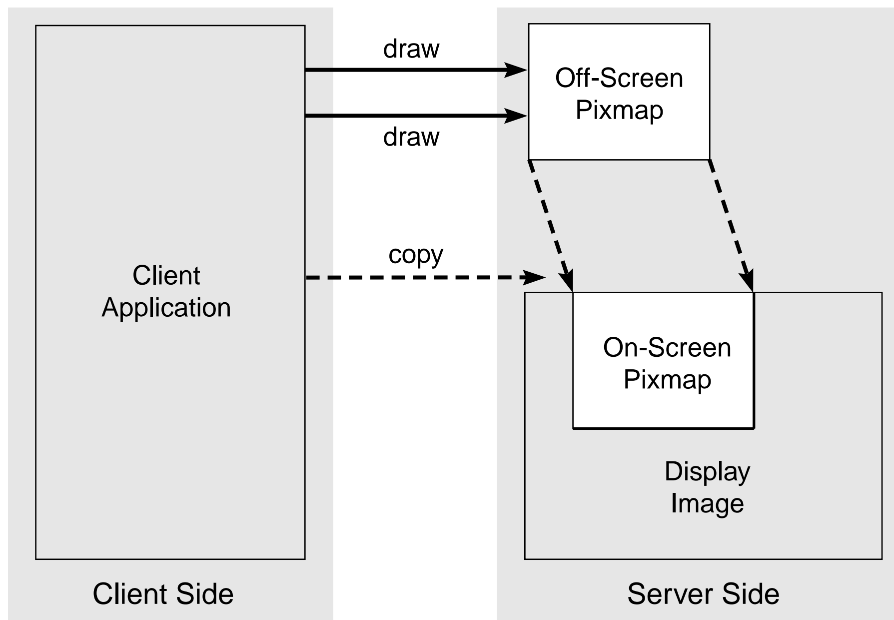


Figure 10.3 Double Buffering with a Pixmap

client can tell it to copy the graphic from the pixmap drawable to the window drawable. Since both of these resources reside on the X server, this will be a fast operation.

A specialized form of the pixmap is the **bitmap,** a 1-bit monochrome pixmap that is commonly used as a stencil or mask. Bitmaps are usually used as a filter for displaying other images and aren't themselves displayed. For example, you might use a bitmap to render an odd-shaped image, copying only the pixels turned on in the bitmap.

10.1.5 Images

Although the pixmap drawable is a good optimization, it still requires that all your drawing commands go across the wire. You may be able to reduce the on-screen flicker, but if you have a lot of drawing commands, you'll still risk bogging down the connection. X provides another mechanism, called an **image,** to allow you to do all your rendering at the client side.

The drawing commands for an image are not nearly as sophisticated as those for a drawable. You are pretty much limited to manipulating the raw

color and pixel data yourself, without the benefit of all the sophisticated line-drawing and image-loading functions that a pixmap drawable enjoys. You also have to know quite a bit about the details of the current visual, and the format in which it expects the image data to be. A different visual may use a different data format, so you may end up writing different code for each color depth.

If you're writing an application in which you need to create and manipulate a local copy of a graphical image, you're probably better off using a higher abstraction (see Section 10.3), and leaving the low-level operations to GDK. You should use an X11 image only if you really know what you're doing.

10.2 The GDK Wrapper

Most of the X concepts we've covered so far apply very closely to GDK, which, you may recall, provides a rather thin wrapper around the Xlib drawing API. We'll touch lightly on the GDK wrapper in this section, before moving on to some actual graphics code in the sections that follow.

10.2.1 Simpler API

The GDK wrapper does a lot to make Xlib more palatable. Possibly the most important thing it does is provide an abstraction layer around Xlib, which makes it easier to port GTK+ and GNOME applications to new graphical systems, like BeOS or Microsoft Windows. If you make sure that your applications don't access anything lower than GDK, you'll increase your chances of an easier port. When GDK is ported to a new system, porting your own application will often be as simple as a recompile and a handful of minor tweaks (depending, of course, on what goes on in your particular application). GDK also remaps the Xlib API into a form more appropriate for and consistent with GTK+ and GNOME. Where possible, it simplifies the parameter lists of each function, making it easier to use.

Let's look at how the Xlib API for creating a new pixmap compares to the GDK API. The relevant Xlib function is `XCreatePixmap( )`:

```
Pixmap XCreatePixmap(Display *display, Drawable d,
    unsigned int width, unsigned int height,
    unsigned int depth)
```

The GDK wrapper function combines the first two parameters of `XCreatePixmap( )` into the single `GdkWindow` parameter and changes the function name into the GDK style: `gdk_pixmap_new( )`. It also converts the `unsigned int` parameters into the more abstracted `gint` types:

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

As we'll see in Section 10.3, GDK also adds quite a bit of new functionality that doesn't exist in Xlib's API. In addition, it provides wrappers for all of the important X11 concepts, such as drawables (GdkDrawable, GdkPixmap, and GdkWindow), colors (GdkColor and GdkColormap), visuals (GdkVisual), and even fonts and cursors (GdkFont and GdkCursor).

10.2.2 Using Color in GDK

Color management can be a convoluted nightmare with raw Xlib applications. At that level you have to keep track of which visual you're in, how the color maps are laid out, whether or not you can create new colors in the color maps, and many other tedious, repetitive details. One of the blessings of GDK is the way in which it abstracts most of these tasks away from the applications programmer. You need to know only a few simple functions, and GDK takes care of the rest.

The first thing you need is the color map. You can get the default system color map with gdk_colormap_get_system( ). If you need to know which color map a specific widget is using, you can call gtk_widget_get_colormap( ) on it. GdkRGB and Imlib (a graphics loading library used by GNOME 1.0) also offer access to their own tuned color maps, with gdk_rgb_get_cmap( ) and gdk_imlib_get_colormap( ), respectively. And finally, if none of these other preexisting color maps are good enough, you can create your own color map with gdk_colormap_new( ), although you will rarely have to go that far. The color map functions look like this:

```
GdkColormap* gdk_colormap_get_system( );
GdkColormap* gtk_widget_get_colormap(GtkWidget *widget);
GdkColormap* gdk_rgb_get_cmap( );
GdkColormap *gdk_imlib_get_colormap( );
GdkColormap* gdk_colormap_new (GdkVisual *visual,
    gint allocate);
```

It is generally a good idea to use the same color map with all widgets in an application; when you start mixing color maps, the color palette may flash whenever the focus switches between widgets with different color maps, especially in lower color depths like 8-bit.

To obtain a color from a color map in GDK, you must allocate it with a call to gdk_colormap_alloc_color( ):

```
gboolean gdk_colormap_alloc_color(GdkColormap *colormap,
    GdkColor *color, gboolean writable, gboolean best_match);
```

The writable parameter tells GDK to add the color to the color map if it's not already there; writable is ignored in read-only color maps. The best_match parameter instructs GDK to find a nearby color if it can't find an exact match, in which case it modifies the color value you passed in to match the color it finds. These two parameters are more or less mutually exclusive: If the color map is writable (such as with GrayScale or PseudoColor visuals), GDK should allocate a new color that *exactly* matches your request rather than trying to find an approximate match. The most common usage is to call the function with a writable of FALSE and a best_match of TRUE.

To specify the color you want, you must populate a GdkColor structure before passing it to gdk_colormap_alloc_color( ). The GdkColor structure is very simple:

```
struct GdkColor
{
  gulong pixel;
  gushort red;
  gushort green;
  gushort blue;
};
```

It's your job to fill in the red, green, and blue fields, and GDK's job to fill in the pixel field with the color map index it finds for that color. You can fill in the RGB values by hand, by choosing 16-bit values for each field, like this:

```
GdkColor red =    { 0, 0xFFFF, 0x0000, 0x0000 };
GdkColor yellow = { 0, 0xFFFF, 0xFFFF, 0x0000 };
GdkColor gray =   { 0, 0x8888, 0x8888, 0x8888 };
```

An easier, more intuitive way to initialize a GdkColor structure is to use the gdk_color_parse( ) function. It accepts a text string with a color name, like "red," "slate gray," or "MediumSpringGreen"; you can also pass in an RGB string—for example, "RGB:FF/FF/00"—to get results similar to those obtained

in the yellow example above. gdk_color_parse( ) invokes the Xlib function XParseColor( ) to look up an RGB value from a table of common colors

```
gint gdk_color_parse(const gchar *spec, GdkColor *color);
```

The master list of these colors is in the rgb.txt file distributed with the X Window System, in the directory /usr/X11R6/lib/X11 or someplace similar.

You then pass the parsed GdkColor to gdk_colormap_alloc_color( ), and if all goes right, you can use that new color. You should always check the return values on both of these functions to make sure that GDK found a valid color for you. Here's a quick example of how to retrieve a medium spring green from the system palette:

```
GdkColor color;
GdkColormap *cmap = gdk_colormap_get_system( );

if (gdk_parse_color("MediumSpringGreen", &color) &&
    gdk_colormap_alloc_color(cmap, &color, FALSE, TRUE))
{
  /* Use our MediumSpringGreen for a nice pastoral scene */
}
else
{
  /* Go find some other nice green */
}
```

Because of the GDK abstraction layer, this simple block of code will work on any visual. You don't have to worry about what the color depth is, or whether or not the color map is writable. GDK will handle these details and return a sensible result or bail out with a FALSE if it can't.

10.3 GdkRGB

The next step up in the drawing hierarchy is the GdkRGB drawing API. Although technically this is still part of GDK, we consider it separately here because unlike most of GDK, which is a thin wrapper, GdkRGB is pure added value and doesn't wrap any specific part of Xlib.

The name "GdkRGB" sounds like it might refer to some sort of object or structure. This is not the case. GdkRGB is a unified set of functions for drawing

on GdkDrawable surfaces—that is, GdkWindow or GdkPixmap. These functions help manage colors for you, behind the scenes. By drawing with GdkRGB, you won't have to deal with visuals and color maps.

10.3.1 The RGB Buffer

GdkRGB stores all of its pixel data in 24-bit color arrays, and then converts the colors and images to the target visual and color map when it renders them to the drawable. Thus it shields you from the hassle of dealing with color lookups and buffer management (remember that each color depth uses a different amount of memory to hold a single on-screen pixel). With GdkRGB, you always work with the same type of 24-bit color array, so you won't have to create special rendering routines for each type of visual.

As the name "GdkRGB" suggests, each pixel contains a red, a green, and a blue element, each one 8 bits in size, implying that each color can have an intensity value ranging from 0 to 255. The values of these three elements are packed consecutively into three guchar elements of the RGB buffer array. You might guess from this arrangement that the size of the RGB buffer is the number of pixels in the image, multiplied by 3. Thus a 25 × 40 image would correspond to a guchar array with 3,000 (25 × 40 × 3) elements. However, this is not the case.

The dimensions of the buffer's image affect the exact size allocated for the buffer. To help optimize access to the later parts of the image buffer, each full row of pixels in the buffer is aligned on a 4-byte (or 4-guchar) boundary. This design typically leads to slightly faster memory lookups at the hardware level, and since graphics rendering consists largely of memory copies in long, repetitive loops, any optimization within these innermost loops can make quite a difference. To keep things simple and customizable, each buffer is assigned a **row stride,** the exact length that each row takes up in the buffer array. The row stride is the sum of all the data bytes in a single row of pixels, plus any extra bytes needed to pad the row to 4-byte alignment.

Let's take our 25 × 40 image as an example. Each 25-pixel row uses 75 bytes of data. To make sure the next row starts on a 4-byte boundary, at 76 bytes, we have to add a single unused byte of padding to the end of each row. The result is that we can calculate the offset to any row by multiplying by 4 rather than by 3,[1] so we have a buffer size of 3,040 (76 × 40) instead of 3,000. Figure 10.4 illustrates the concept of row strides.

It's not easy to aesthetically convert a 24-bit color buffer into a 16-bit color buffer, and it is even harder to convert it into an 8-bit (or lower) color buffer.

1. The advantage in multiplying by 4 is that it is equivalent to a quick, simple 2-bit shift to the left, whereas multiplying by 3 cannot be as easily optimized.

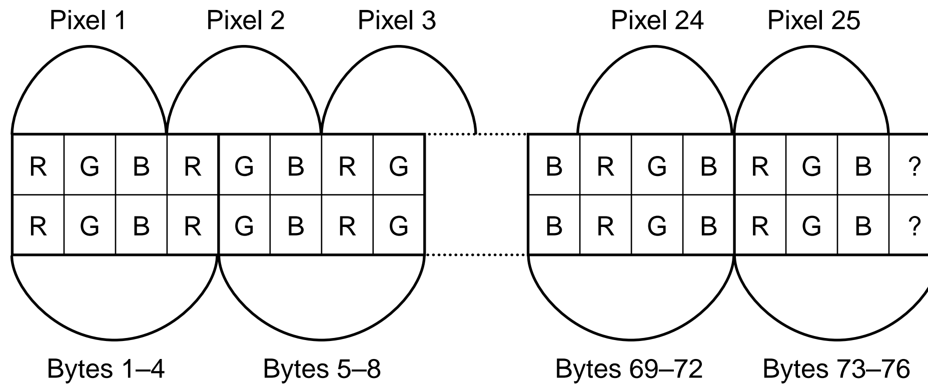


Figure 10.4 RGB Row Strides

The quality of your results will vary wildly, depending on the intricacies of the algorithms you use. For each pixel in your 24-bit image, you will have to search the target visual's color map for an appropriate color. The simplest, and probably fastest, approach is to find the nearest match. When going from 24-bit to 16-bit, you won't lose much quality because you still have 64K colors to choose from.

The real problems come when you try to degrade your 16-million-color image down to 256 colors. Simple closest-match algorithms can't do justice to the image and will lead to severe **banding.** This sharp line of transition between color hues shows up most often in conversions of a smooth color gradient to a lower color depth.

A more sophisticated color conversion algorithm can smooth out this abrupt transition by **dithering** colors. Dithering introduces an element of fuzziness to color conversions. Rather than taking into account only one pixel at a time, a dithering algorithm looks at surrounding pixels and tries to come up with a combination of adjacent pixel colors that the human eye will mistake for an intermediate color. A common application of dithering (especially in newspapers) is the fine-grained black-and-white checkerboard, which appears from a distance to be a flat gray. Sophisticated dithering techniques can do wonders to maintain image quality in conversions to a lower color depth.

This discussion about color conversions is intended to illustrate the amount of work that goes on behind the scenes in GdkRGB. If it weren't for GdkRGB, you would have to implement all these conversion routines yourself to produce consistent-looking images across the full range of color depths and

visuals. By analogy, if you've never bicycled from Detroit to Albuquerque, you may not appreciate how much more convenient it is to fly there. GdkRGB makes the arduous journey from one color depth to another quick and easy.

If you wish to use GdkRGB in your application, you will need to initialize it, as shown here, before you call any of its functions:

```
void gdk_rgb_init( );
```

GdkRGB makes use of global variables that need to be loaded in order for it to function properly. Failure to initialize GdkRGB can result in application crashes, so don't forget this step! If you are using GdkRGB indirectly, through a wrapper such as GNOME or gdk-pixbuf (see Section 10.5), you shouldn't have to initialize it, because the wrapper will take care of that for you.

If you get into trouble and need help figuring out why your calls to GdkRGB aren't doing what you think they should be doing, you can turn on the `verbose` debugging mode by calling the following function with a value of `TRUE`:

```
void gdk_rgb_set_verbose(gboolean verbose);
```

You can turn it back off by calling the function again with a `FALSE`. It may not tell you exactly what's going on, but if you're lucky, it will give you a hint or two.

10.3.2 Drawing Functions

The most impressive feature of the GdkRGB API is its set of drawing routines, each of which targets a `GdkDrawable` instance. The drawing routines are quite similar, differing primarily in the particular algorithms they use to transfer and downgrade the 24-bit colors into the target drawable.

In most cases you will be able to get by with using only the first drawing function, `gdk_draw_rgb_image( )`. This function, like the others, copies a rectangular block of pixels from anywhere inside the RGB buffer into the target drawable, automatically converting it to the proper visual. You must also supply the dithering style, a pointer to the RGB buffer, and your buffer's row stride so that GdkRGB knows exactly where to wrap the pixel rows:

```
Void gdk_draw_rgb_image(GdkDrawable *drawable,
    GdkGC *gc, gint x, gint y, gint width, gint height,
    GdkRgbDither dith, guchar *rgb_buf, gint rowstride);
```

The dith parameter can be GDK_RGB_DITHER_NONE for no dithering (remember our discussion about banding), GDK_RGB_DITHER_NORMAL to tell GdkRGB to dither only when rendering to 8-bit color visuals, or GDK_RGB_DITHER_MAX to dither on both 8-bit and 16-bit visuals. Usually the reduction in performance that accompanies dithering on a 16-bit color drawable isn't worth the slight improvement in appearance, so your best bet is GDK_RGB_DITHER_NORMAL. The gc parameter points to a GdkGC structure, which should hold any contextual information for the drawing operation (see Section 2.2.4).

GdkRGB has four additional specializations of the basic gdk_draw_rgb_image( ) function. The first two constrain the rendering to a specific style of visual. gdk_draw_gray_image( ) uses only shades of gray to render to the drawable, regardless of the target visual. You can use gdk_draw_indexed_image( ) to force GdkRGB to render according to a color map you specify. Rather than using the native GDK API for creating and populating your color map, GdkRGB provides an easy-to-use convenience function, gdk_rgb_cmap_new( ). This color map is allocated dynamically, so you must free it when you're done, using gdk_rgb_cmap_free( ):

```
void gdk_draw_gray_image(GdkDrawable *drawable,
    GdkGC *gc, gint x, gint y, gint width, gint height,
    GdkRgbDither dith, guchar *buf, gint rowstride);
void gdk_draw_indexed_image (GdkDrawable *drawable,
    GdkGC *gc, gint x, gint y, gint width, gint height,
    GdkRgbDither dith, guchar *buf, gint rowstride,
    GdkRgbCmap *cmap);
GdkRgbCmap* gdk_rgb_cmap_new (guint32 *colors, gint n_colors);
void gdk_rgb_cmap_free (GdkRgbCmap *cmap);
```

The other two drawing functions have a less visible effect on the image. The gdk_draw_rgb_image_dithalign( ) function is good for rendering parts of a dithered image, particularly to a scrollable drawing surface. In order for the dithering algorithms to match up seamlessly, without leaving noticeable transition lines, you must supply two extra parameters—xdith and ydith—the offset of the scrolled image from its base, unscrolled position. Supplying consistent values of xdith and ydith ensures that multiple passes of the dithering algorithm will generate the same pattern, even when the rendering region shifts.

The final drawing function, gdk_draw_rgb_32_image( ), is more of a friendly curiosity than a critical supplement. It attempts to align every 3-byte

color value in the RGB buffer along a 4-byte boundary to improve memory access time. Rather than just aligning the final pixel of a row, it aligns every pixel in the image. In practice, however, the added speed is not enough to justify the significant increase in memory consumption.

```
void gdk_draw_rgb_image_dithalign(GdkDrawable *drawable,
    GdkGC *gc, gint x, gint y, gint width, gint height,
    GdkRgbDither dith, guchar *rgb_buf, gint rowstride,
    gint xdith, gint ydith);
void gdk_draw_rgb_32_image(GdkDrawable *drawable,
    GdkGC *gc, gint x, gint y, gint width, gint height,
    GdkRgbDither dith, guchar *buf, gint rowstride);
```

10.3.3 Color Management

As we learned in Section 10.1, color management can be a tricky process at the lower levels. One of the most tedious tasks is setting up the visual and the color map for a given display. A simple mistake inside the complex color management code can cause your application to crash and bail out when it becomes impossible to render to a visual in a sane way, leading to X's nebulous "Bad Match" errors.

Another helpful abstraction service that GdkRGB provides is the automatic generation of a rational visual and color map for whichever display the application runs on. If one user runs the application on an 8-bit pseudocolor display, GdkRGB detects that and creates an appropriate indexed color map. If another user runs the same application on a 24-bit color display, GdkRGB creates a TrueColor color map and its corresponding visual instead. You can access this customized color map and visual at any time after `gdk_rgb_init()` has run, by calling these two functions:

```
GdkColormap* gdk_rgb_get_cmap();
GdkVisual* gdk_rgb_get_visual();
```

You can force a widget to use GdkRGB's color map and visual by pushing it onto a pair of global stacks (really a pair of singly linked lists used to simulate the stacks) in GTK+ and then popping it back off when you're done. You need to push it only for the time it takes to create the widget. Once created, the widget will continue to use that color map and visual for the rest of its life span; all of the widget's children will inherit them too. The common practice is to push them, create the top-level widget, and then pop them immediately afterward:

```
gtk_widget_push_visual(gdk_rgb_get_visual( ));
gtk_widget_push_colormap(gdk_rgb_get_cmap( ));
widget = gtk_widget_new(...);
gtk_widget_pop_visual( );
gtk_widget_pop_colormap( );
```

GdkRGB also supplies a convenience function for looking up values in this `GdkColormap`. If you feed it a 24-bit RGB color value, it will return that color's corresponding value in the color map. You can then use the returned color in the various raw GDK functions. Two other functions allow you to set the default foreground and background colors of the graphics context with RGB colors, rather than having to perform the lookup first, as you would normally have to do with the `gdk_gc_set_fore/background( )` functions.

```
gulong gdk_rgb_xpixel_from_rgb(guint32 rgb);
void gdk_rgb_gc_set_foreground(GdkGC *gc, guint32 rgb);
void gdk_rgb_gc_set_background(GdkGC *gc, guint32 rgb);
```

Finally, you can use `gdk_rgb_ditherable( )` to find out whether or not GdkRGB's visual is ditherable, and act accordingly. You might use this function to decide whether to call `gdk_draw_rgb_image( )` or `gdk_draw_rgb_image_ditherable( ):`

```
gboolean gdk_rgb_ditherable( );
```

10.4 Libart

Libart brings a very important feature to GNOME: high-quality anti-aliased, vector-based image manipulation. Libart's assortment of paths, points, and regions gives you very fine-grained control over important tasks, such as image rotation, scaling, and shearing. Libart is optimized for quick, efficient rendering, making it a prime candidate for graphic-intensive applications.

A full presentation of the libart API could easily take up an entire chapter or two, so to keep things short we will focus on the basic concepts and data structures, with the hope that you can use this knowledge to explore and learn the finer details of this library more quickly on your own.

10.4.1 Vector Paths

A **vector path** in libart is an array of coordinates that form a consecutive series of line segments, connected end to end. A path can be open, as with a line graph, or it can be closed, as with a polygon or figure eight. A path can cross over itself any number of times. The structure that libart uses to keep track of each point in a vector path looks like this:

```
typedef struct _ArtVpath ArtVpath;
struct _ArtVpath
{
  ArtPathcode code;
  double x;
  double y;
};
```

Figure 10.5 shows an array of five `ArtVpath` elements tracing the four sides of a diamond shape.

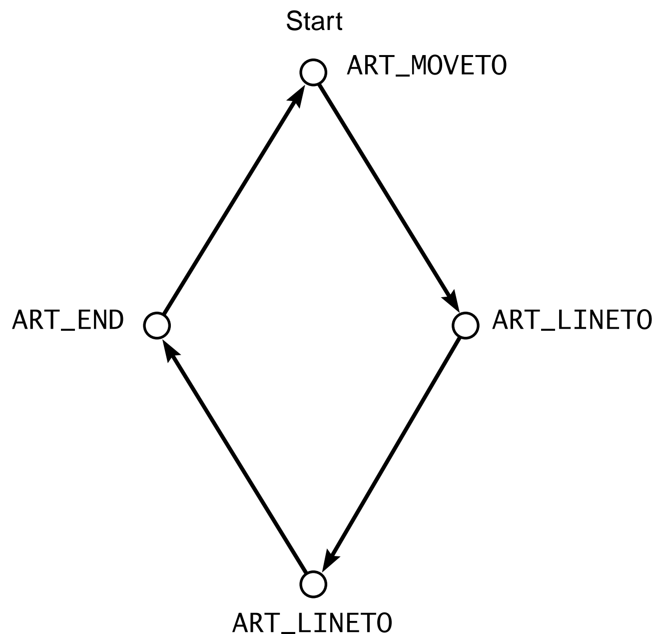


Figure 10.5 Array of `ArtVpath` Elements

Each element consists of a single coordinate point and one property, the ArtPathcode enumeration, which tells libart how this point connects to the previous point, if any:

```
typedef enum
{
  ART_MOVETO,
  ART_MOVETO_OPEN,
  ART_CURVETO,
  ART_LINETO,
  ART_END
} ArtPathcode;
```

The first point in a path array must be ART_MOVETO if the path is a closed path or ART_MOVETO_OPEN for an open path. The final point is always ART_END. All other points in a vector path array must be ART_LINETO. In a closed path, libart will implicitly connect the ART_END point to the initial ART_MOVETO point.

By definition, a vector path consists of all straight lines. For this reason, ART_CURVETO is not permitted in vector paths. As we will see in the next section, the more complex Bézier paths can handle curves.

10.4.2 Bézier Paths

When you can get away with only straight lines, use vector paths. The calculations are simple and fast, and the paths are easy to build. However, if you are trying to draw anything natural-looking, straight lines will make it look blocky and clumsy. At some point you will probably want to render a real curve. This is where **Bézier paths** come into play.

Each element of a Bézier path requires three coordinate points rather than the single point used by a vector path element. In a nutshell, the Bézier curve is defined by its starting and ending points, as well as a middle point, usually off to the side. The curve will trace a path from the first coordinate to the final coordinate, pulled away from a straight line by the middle coordinate in a predictable, mathematical curve. This path is defined by the following cubic equations, where *t* goes from 0 to 1:

$$x = x_0(1-t)^3 + x_1(1-t)^2 t + x_2(1-t)t^2 + x_3 t^3$$

$$y = y_0(1-t)^3 + y_1(1-t)^2 t + y_2(1-t)t^2 + y_3 t^3$$

Libart uses the following data structure to hold the data for each Bézier line segment; a Bézier path is an array of one or more of these segments. It can be open or closed, and it can be a mixture of straight lines (where only the x_3 and y_3 coordinates are used) and curved lines.

```
typedef struct _ArtBpath ArtBpath;
struct _ArtBpath
{
  ArtPathcode code;
  double x1;
  double y1;
  double x2;
  double y2;
  double x3;
  double y3;
};
```

10.4.3 Sorted Vector Paths

As part of its suite of optimizations, libart defines a **sorted vector path** (**SVP**) to organize a normal vector path into groups of similar segments. Each SVP element consists of two or more points (i.e., a vector path) tracing a path of coordinates that steadily increases or decreases along the *y*-axis. This rule helps optimize algorithms that render in a vertical direction, in particular those used to display image buffers on the screen.

Consider a path shaped like the letter V. A vector path would describe it with a downward (increasing *y*-axis) stroke, followed by an upward (decreasing *y*-axis) stroke. A sorted vector path would break it up into two downward strokes converging on the same point, thus making it easier to render the V from the top down. Figure 10.6 shows the letters V and R broken down into vector paths and sorted vector paths.

Another important feature of SVPs that helps optimize their rendering code is the **bounding box** assigned to each SVP segment. This bounding box defines the minimum rectangular area that can contain that SVP segment (which, remember, can contain more than two points, like the second segment in the "R" example). When part or all of a sorted vector path needs to be redrawn, the area that must be "dirtied" can be clipped and reduced quite a bit, so that only the areas covered by those bounding boxes are repainted rather than a single, huge rectangle that covers the entire path. This clipping can save a lot of needless work.

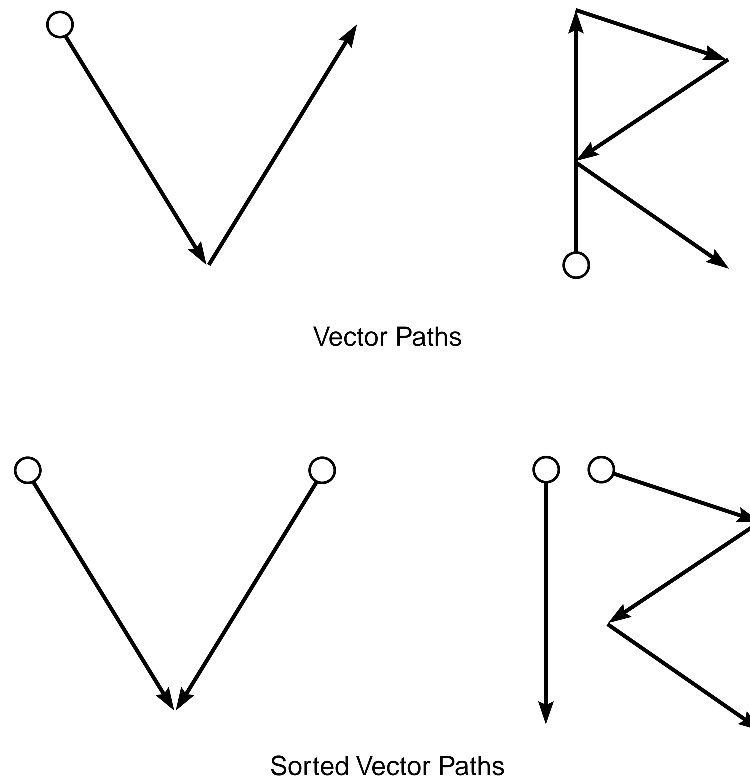


Figure 10.6 Vector Paths of the Letters V and R

10.4.4 Microtile Arrays

Libart's microtile arrays make it easier to optimize image redrawing, by breaking the "dirty" areas down into smaller rectangles. The microtile array (affectionately abbreviated uta, with the "u" standing for the Greek letter μ, or micron) represents a grid of 32 × 32 pixel squares overlaid on an area of coordinate space. For example, a 96 × 64 pixel buffer would have six microtiles, arranged in a 3 × 2 grid.

Figure 10.7 shows a path drawn in the shape of a guitar case. Rather than having to rerender the bounding box of the entire path, indicated by the dotted line, the microtile array breaks the dirty regions into smaller components. The gray rectangles show the areas that need to be refreshed when using the uta.

Each microtile has a single bounding box within its 32 × 32 area, to mark the entire area of interest—the "dirty" area in the case of repainting

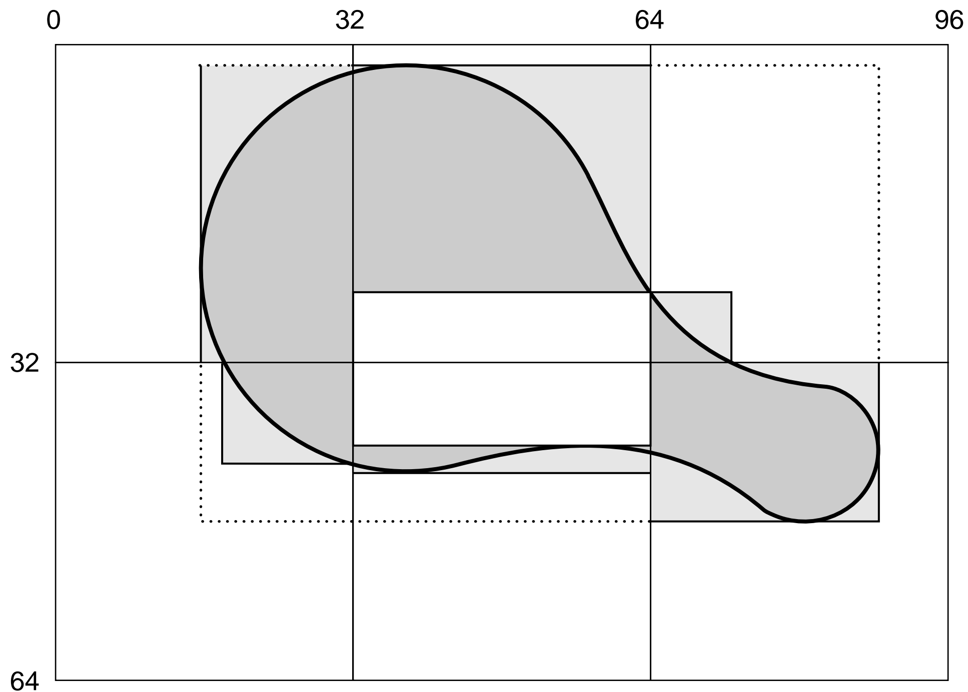


Figure 10.7 Microtile Arrays

algorithms—within its grid square. When an application attempts to refresh its microtiled drawing areas, it can roll through the uta and redraw only the area contents of each bounding box. It can ignore all microtiles with zero-area bounding boxes. The fewer unnecessary pixels the application has to redraw, the faster the refresh will take place each time. In practice, the microtiles are usually merged and decomposed into an array of rectangles, optimized for efficient redraws.

A significant advantage to utas is that you can add to them dynamically, as the dirty area changes. The microtiles will adjust their bounding boxes to include the new areas or ignore the areas they already cover. This makes it much easier to implement deferred painting updates, in which redraws happen during idle times. Between updates, the microtiles accumulate "dirt," but they always remain in a ready-to-use state, whenever the next repaint event happens to occur.

Another advantage of microtile arrays is their stable memory footprint. No matter how many times a drawing area is damaged, the number of microtiles

never increases. Only the coordinates of the existing bounding boxes change. Utas are also very efficient, especially because of their 32-pixel grid size. The 32-pixel boundaries result in convenient 4-byte-aligned chunks of the pixel buffer. Also, the fact that each bounding box is constrained to a coordinate range of 0 to 32 pixels makes it possible to fit all four coordinate values into a single 32-bit integer. Technically, each coordinate fits into 5 bits, but to maintain the optimization, each coordinate is padded to 8 bits. Thus our earlier 96×64 pixel example requires only 24 bytes (6 microtiles $\times$ 4 bytes) to express its bounding-box data.

10.4.5 Affine Transformations

Now that we have the basic building blocks of a vector drawing kit, we need some way to move those vectors around. At the very least, we should be able to move them through space, or **translate** them. Another important feature is **scaling** them to larger or smaller sizes, to simulate zooming and unzooming. Even more sophisticated is the ability to **rotate** our lines and polygons. Finally, we might like to be able to **shear** our vector paths—for example, to turn a normal rectangle into a slanted parallelogram.

Libart implements these operations as **affine transforms;** each transform is characterized by an array—or matrix—of six floating-point numbers. To apply the affine transform on a path, libart passes each coordinate pair through these two operations, where x and y are the old coordinates, x' and y' are the new coordinates, and affine_0 through affine_5 make up the array of floating-point numbers:

$$x' = \text{affine}_0 x + \text{affine}_2 y + \text{affine}_4$$

$$y' = \text{affine}_1 x + \text{affine}_3 y + \text{affine}_5$$

Surprisingly, these innocent-appearing little functions can result in quite a number of dazzling effects, in particular the four transforms already mentioned. By carefully tweaking the affine matrix, we can invent many strange new transforms. Of particular note is the simplicity of the calculations. Affine transforms involve only multiplication and addition, two fairly quick operations that are not likely to bog down your rendering routines. You don't have to worry about any complicated square roots or cubic equations sneaking into your back-end algorithms.

The simplest transform of all is **identity,** which leaves all points in the path unaltered. The identity looks like (1, 0, 0, 1, 0, 0) and simplifies to the following equations:

$$x' = 1x + 0y + 0 = x$$

$$y' = 0x + 1y + 0 = y$$

A translation transform is a simple offset, which looks like (1, 0, 0, 1, *m, n*), in which every point is moved *m* units horizontally and *n* units vertically:

$$x' = 1x + 0y + m = x + m$$

$$y' = 0x + 1y + n = y + n$$

Scaling is similar to the identity transform, except it uses values greater or less than 1 and looks like (*m,* 0, 0, *n,* 0, 0). Values greater than 1 will enlarge, or zoom in, and values between 0 and 1 will shrink, or zoom out. Negative values will flip the path to the opposite side of the axis, which is not normally what you want, so you'll probably want to keep your multipliers in the positive range:

$$x' = mx + 0y + 0 = mx$$

$$y' = Ox + \mathrm{ny} + 0 = ny$$

Rotation and shearing transforms are a little more complex, and they usually involve some sort of trigonometry. To see how they work, take a look at libart's implementation of them. The source code is in the gnome-libs package in the file `gnome-libs/libart_lgpl/art_affine.c`, located in the `art_affine_rotate( )` and `art_affine_shear( )` functions. The other mentioned transforms reside in `art_affine_identity( )`, `art_affine_translate( )`, and `art_affine_scale( )`. You'll find other inter-

esting transforms in `art_affine_invert()`, `art_affine_flip()`, and `art_affine_point()`; you can compare two transforms with `art_affine_equal()` and combine two transforms with `art_affine_multiply()`. If you use only one thing in libart, it will most likely be the affine transforms, so it's a good idea to figure out how they work. They also pop up from time to time in the GNOME Canvas (see Chapter 11).

Affine transformations have some very interesting properties that make their simplicity even more attractive. First, affines are linear, so they always transform straight lines into straight lines; also, if two lines are parallel, they will remain parallel after any number of transforms. Closed paths will remain closed.

Another interesting property is that affine transforms are cumulative. You can combine multiple transforms into a single composite transform with `art_affine_multiply()`, so if you have a series of transformations that you always perform in order, you can speed things up by combining them into a single transform. The results will be exactly the same. Note, however, that order is still important. If you exchange steps 3 and 4 in a six-step series, you will very likely come out with different results (depending on the specifics of the transforms; obviously if steps 3 and 4 were both identity transforms, it wouldn't matter).

Finally, affine transforms are mathematically reversible. You can apply a transform, invert it with `art_affine_invert()`, and apply the inversion, and you will end up with your original path (taking into account any rounding errors).

10.4.6 Pixel Buffers

One of the highlights of libart is its support for RGB and RGBA (red-green-blue-alpha) pixel buffers, or **pixbufs.** All of the various data constructs and algorithms come to fruition in these buffers. You can draw lines and shapes on the pixel buffer by creating an SVP (or by creating a vector path and converting that into an SVP) and rendering it to the pixbuf with a stroking operation. You can apply any number of affine transforms on images to offset, scale, and otherwise morph them. Libart also provides pixel buffer life cycle management in the form of creation and destruction functions.

Libart defines its own data structure, `ArtPixBuf`, to hold the important elements:

```
typedef struct _ArtPixBuf ArtPixBuf;
struct _ArtPixBuf
```

```
{
  ArtPixFormat format;
  int n_channels;
  int has_alpha;
  int bits_per_sample;

  art_u8 *pixels;
  int width;
  int height;
  int rowstride;
  void *destroy_data;
  ArtDestroyNotify destroy;
};
```

Most of the fields in `ArtPixBuf` should look familiar from our discussions in the earlier parts of this chapter. Only a few of them represent new concepts.

The `format` field holds the structural format of the pixel buffer. For now, the `ArtPixFormat` enumeration allows only one value, `ART_PIX_RGB`, signifying a three- or four-channel RGB buffer. In the future, libart might support other color formats, such as grayscale, CMYK (cyan-magenta-yellow-black), and so on.

The `n_channels` field determines the number of color channels the pixel buffer has. A three-color RGB buffer has three channels; a three-color RGB pixel buffer with an alpha channel has an `n_channels` of 4. All other values are invalid for `ART_PIX_RGB`.

`ArtPixBuf` explicitly states whether or not one of its channels is an alpha channel with the `has_alpha` field. Although you can probably figure this out by looking at `n_channels`, you should avoid doing so. Other pixbuf formats may have a different number of base channels. For example, grayscale might have one color channel and one alpha channel, while CMYK might have four color channels and one alpha channel. The `has_alpha` field is the only surefire way to test for the presence of an alpha channel.

The `bits_per_sample` field is another way of expressing the color depth of the pixel buffer. A sample is analogous to a color channel, so we can calculate the bit depth of a buffer by multiplying `n_channels` by `bits_per_sample`. Normally this value stays at 8, implying 24-bit color depth for a three-channel RGB pixbuf.

The next four fields describe the structure of the buffer. The `pixels` field points to the actual buffer data, which is simply an array of `unsigned char` (defined by `typedef` to `art_u8`). The `width`, `height`, and `rowstride` fields

control how the `pixels` array is divided into rows and columns, with potential padding bytes at the end of each row, as indicated by `rowstride`.

The final two fields, `destroy_data` and `destroy`, are optional fields that permit you to register a callback function that libart will invoke just before it destroys the `ArtPixBuf` instance. You don't need to register the `destroy` callback unless you have data of your own that you'd like automatically cleaned up when the owning `ArtPixBuf` instance is destroyed. The `ArtDestroyNotify` function prototype looks like this:

```
typedef void (*ArtDestroyNotify) (void *func_data, void *data);
```

Libart provides several functions to automatically fill in these fields with the proper values. You use a different function to create a three-channel pixel buffer than you do to create a four-channel buffer, and yet another set of functions if you want to register a `destroy` callback. Here's a quick listing of the commonly used `ArtPixBuf` creation and destruction functions:

```
ArtPixBuf* art_pixbuf_new_rgb(art_u8 *pixels,
    int width, int height, int rowstride);
ArtPixBuf* art_pixbuf_new_rgba(art_u8 *pixels,
    int width, int height, int rowstride);
ArtPixBuf* art_pixbuf_new_rgb_dnotify(art_u8 *pixels,
    int width, int height, int rowstride,
    void *dfunc_data, ArtDestroyNotify dfunc);
ArtPixBuf* art_pixbuf_new_rgba_dnotify(art_u8 *pixels,
    int width, int height, int rowstride,
    void *dfunc_data, ArtDestroyNotify dfunc);
void art_pixbuf_free (ArtPixBuf *pixbuf);
```

You may have noticed that `ArtPixBuf` has a lot of fields similar to those found in the GdkRGB API. It even seems to implement a fair amount of the same functionality. Although at first this may seem like a duplication of effort,[2] the two APIs are kept separate for good reason. GdkRGB is intimately tied to GDK, and by extension to GLib. Most of GdkRGB's focus is on rendering pixel buffers to GDK drawables, performing dithering, and managing color maps—all operations that are very specific to windowing systems.

On the other hand, libart is very generic, with as few external dependencies as possible. For this reason libart can survive outside of GTK+ and

2. It certainly shouldn't be, since GdkRGB and libart were created by the same author, Raph Levien.

GNOME; it doesn't even need GLib. Libart is free from all windowing-system and GUI constraints. In the true UNIX spirit, it does one thing and it does it very well: pixel buffer manipulation. Thus if you want to make use of libart, you will have to provide your own code to transfer the contents of the ArtPixBuf structures to a drawing widget of some sort. Usually the easiest way is to transfer your pixel buffers to a G+kDrawingArea widget, using GdkRGB. The GNOME Canvas implements a similar approach in its anti-aliased mode, using GdkRGB and the Canvas widget (see Chapter 11).

10.5 Gdk-pixbuf

Libart is designed to manipulate graphics buffers, but it contains no functions to help load existing images into those buffers. It leaves the buffer creation and loading up to you. The gdk-pixbuf library is a toolkit for image loading and pixel buffer manipulation that you can use in conjunction with libart. Gdk-pixbuf also provides convenience functions for progressive image loading, animation (which we won't cover here), and rendering the libart image buffer to a GdkDrawable instance.

Gdk-pixbuf has a fairly large API. We'll discuss it briefly as we go, and then we'll wrap up the chapter with a sample application. The fundamental currency in the gdk-pixbuf library is the GdkPixbuf structure, a private, opaque data structure that mirrors many of the same concepts that ArtPixBuf supports. In fact, most of GdkPixbuf's private data fields have the same names and data types as the corresponding ones in ArtPixBuf. This similarity dates back to the earlier days when gdk-pixbuf was a wrapper around libart. Since that time, the libart dependency has been stripped out, and gdk-pixbuf is now scheduled to be merged into the GTK+ 2.0 code base. As such, gdk-pixbuf's days as a standalone library are limited to the GNOME 1.x release.

10.5.1 Creating

To get started, you will have to create a new GdkPixbuf instance. If you're planning on populating the RGB buffer from scratch, or if you want to load several smaller pixel buffers into a larger one, you may want to use gdk_pixbuf_new( ), which creates an uninitialized pixel buffer of guchar elements:

```
GdkPixbuf *gdk_pixbuf_new(GdkColorspace colorspace,
    gboolean has_alpha, int bits_per_sample,
    int width, int height);
```

The guchar type is essentially the same data type as libart's art_u8 type, so you can use libart to manipulate the raw pixel buffers in gdk-pixbuf. Given gdk-pixbuf's history with libart, this should come as no surprise.

The first parameter is a value in the GdkColorspace enumeration, analogous to libart's ArtPixFormat enumeration. For now, you can specify only the value GDK_COLORSPACE_RGB, but other types of color spaces may be supported in the future:

```
typedef enum
{
  GDK_COLORSPACE_RGB
} GdkColorspace;
```

You can request an optional alpha channel for determining image opacity with the has_alpha parameter. The bits_per_sample parameter behaves just like the bits_per_sample field in the ArtPixBuf structure. For the time being, you should always pass in an 8. In the future, gdk-pixbuf might support additional channel depths, but currently it supports only 8-bit channels (just like libart). Finally, you must pass in the width and height of your desired buffer; gdk-pixbuf uses these parameters to calculate the best possible row stride for the new buffer.

After you've created and initialized your GdkPixbuf structure, you can retrieve information about it with the following set of accessor functions:

```
GdkColorspace gdk_pixbuf_get_colorspace(
    const GdkPixbuf *pixbuf);
int gdk_pixbuf_get_n_channels(const GdkPixbuf *pixbuf);
gboolean gdk_pixbuf_get_has_alpha(const GdkPixbuf *pixbuf);
int gdk_pixbuf_get_bits_per_sample(const GdkPixbuf *pixbuf);
guchar *gdk_pixbuf_get_pixels(const GdkPixbuf *pixbuf);
int gdk_pixbuf_get_width(const GdkPixbuf *pixbuf);
int gdk_pixbuf_get_height(const GdkPixbuf *pixbuf);
int gdk_pixbuf_get_rowstride(const GdkPixbuf *pixbuf);
```

You can also create a GdkPixbuf structure from an existing image or buffer. One approach is to load it up with a graphics file. You can give the gdk_pixbuf_new_from_file() function the path to a graphics file. Gdk-pixbuf will automatically invoke one of a handful of its image format loaders to convert the graphics file into an RGB buffer. A second function, gdk_pixbuf_new_from_xpm_data(), loads inline XPM data:

```
GdkPixbuf *gdk_pixbuf_new_from_file(const char *filename);
GdkPixbuf *gdk_pixbuf_new_from_xpm_data(const char **data);
```

Gdk-pixbuf currently supports PNG, XPM, JPEG, TIFF, PNM, RAS, BMP, and even the patented GIF format. The loaders are easy to write and very modular, so gdk-pixbuf is well equipped to keep pace with new and old image formats alike.

Gdk-pixbuf even lets you create a pixel buffer from an existing raw data buffer. The `gdk_pixbuf_new_from_data( )` function is similar to `gdk_pixbuf_new( )`, except it has four extra parameters: `data` to pass in your RGB buffer, `destroy_fn` and `destroy_fn_data` to register a callback function to clean up the `data` buffer when the `GdkPixbuf` is destroyed, and `rowstride`:

```
typedef void (* GdkPixbufDestroyNotify) (guchar *pixels,
    gpointer data);
GdkPixbuf *gdk_pixbuf_new_from_data(const guchar *data,
    GdkColorspace colorspace, gboolean has_alpha,
    int bits_per_sample, int width, int height, int rowstride,
    GdkPixbufDestroyNotify destroy_fn,
    gpointer destroy_fn_data);
```

You must calculate and pass in a rational row stride for the pixel buffer you've created. The pixel buffer is really only a one-dimensional array of `guchar` elements, so gdk-pixbuf can't guess from the `height` and `width` parameters what your intended row stride should be. In the case of `gdk_pixbuf_new( )`, gdk-pixbuf creates its own pixel buffer and can make its own decision about the row stride.

You should use this function only if you know what you're doing. Gdk-pixbuf does very little safety checking on the imported buffer, so if your `rowstride` value doesn't properly match the rows of pixels in your buffer, your image will end up scrambled. Gdk-pixbuf unquestioningly accepts your values, trusting you not to make any mistakes.

Sometimes you end up with a three-channel RGB pixel buffer that you later need to convert into a four-channel RGBA buffer. The most common example of such a situation occurs when you load a three-channel image from a file but need to use the alpha channel for masking. The file-loading modules create whichever style of `GdkPixbuf` best matches the image file. You can't tell it ahead of time that you need four-channel pixel buffers, so you're forced to convert it to RGBA after the fact.

Although it may seem simple at first to tack the extra channel onto each pixel, you'll find that things are a little more complicated if you try this by hand. First you have to create a new `GdkPixbuf` and copy the pixels into it one by one, adding the extra alpha channel byte to each one as you go. An added twist is the fact that your row stride will always change, so the number of unused padding bytes at the end of each row will change as well. You can't just do a simple byte-by-byte transfer.

Recall our row stride example from Section 10.3.1, in which an image with a width of 25 pixels had a row stride of 76 ($25 \times 3 = 75$), which amounts to a single byte of padding per row. Adding an alpha channel to this image will result in a row stride of 100 (25×4), which means that the new buffer will have no padding bytes. Every time you copy over a row of pixels, you will have to drop a byte.

To save you the effort of reimplementing this algorithm in each application that needs it, gdk-pixbuf offers the `gdk_pixbuf_add_alpha( )` convenience function. Simply pass in your three-channel pixel buffer, and this function will generate a new four-channel copy. If you want, you can preload the new alpha channel on the basis of the contents of the image. Set the `substitute_color` parameter to `TRUE` and fill the `r`, `g`, and `b` parameters with target color values. The function will turn on the corresponding pixels in the alpha channel each time it finds the target color. This feature is great for transparency, when the backgrounds of your three-channel images have been set to a special masking color. Conversely, if you set `substitute_color` to `FALSE`, the function will ignore the three color parameters and initialize the alpha channel to show the full image:

```
GdkPixbuf *gdk_pixbuf_add_alpha(const GdkPixbuf *pixbuf,
    gboolean substitute_color, guchar r, guchar g, guchar b);
```

To make it easier to share and manage RGB buffers, gdk-pixbuf implements the familiar reference-counting scheme (see Section 2.2.5). All newly created pixbufs start out with a reference count of 1, so you should need to call `gdk_pixbuf_ref( )` for only the second and later owners. When you're done with the pixbuf, always call `gdk_pixbuf_unref( )`, which automatically destroys the pixbuf when the reference count reaches 0, but not before. For this reason gdk-pixbuf doesn't have an explicit destroy function.

```
void gdk_pixbuf_ref(GdkPixbuf *pixbuf);
void gdk_pixbuf_unref(GdkPixbuf *pixbuf);
```

If you need a physically separate copy of a pixel buffer, and not just a shared reference to an existing one, you can use gdk_pixbuf_copy( ) to create a new pixbuf identical to the one you pass in:

```
GdkPixbuf* gdk_pixbuf_copy(const GdkPixbuf *pixbuf);
```

10.5.2 Rendering

After you have tweaked your image buffer and loaded it into a GdkPixbuf structure, you'll probably want to display it on the screen. GNOME offers various ways to do this, particularly within the GnomeCanvas widget (see Section 11.4.6). In this chapter we'll focus on displaying the buffer in a GdkDrawable instance.

One of the most frustrating limitations of the current incarnation of the X Window System is its lack of support for a full-fledged alpha channel. In X you cannot perform all the alpha blending that libart and gdk-pixbuf support. You can only use alpha blending and anti-aliasing on local image buffers, and then transfer those buffers as a whole to the X drawables. There are tentative plans for an alpha channel extension to X, but those hopes are off in the future.

The basic rendering tools of X support only binary masks: Either copy a pixel in its original color, or don't copy it at all. X does not allow you to perform color blending during an image copy. You can render shaped images to a drawable by specifying a 1-bit mask (a GdkBitmap), but those images will have sharp edges. You won't be able to smoothly blend your new image into the old image residing in the GdkDrawable. Thus if you want anti-aliased drop shadows, you must take care of them within the RGB buffer, *before* you render the image to the X drawable. Internally, gdk-pixbuf uses GdkRGB to handle the complexities of color maps, visuals, and dithering.

To express the nature of the mask it will use when rendering to a drawable, gdk-pixbuf declares the GdkPixbufAlphaMode enumeration:

```
typedef enum
{
  GDK_PIXBUF_ALPHA_BILEVEL,
  GDK_PIXBUF_ALPHA_FULL
} GdkPixbufAlphaMode;
```

For the time being, for the reasons already stated, you must always use GDK_PIXBUF_ALPHA_BILEVEL. This value tells gdk-pixbuf to convert the pixel buffer's alpha channel (if any) into a 1-bit bilevel mask. When full alpha blend-

ing finally becomes part of the X Window System, you can start using GDK_PIXBUF_ALPHA_FULL.

So, to render a nonrectangular shape to a drawable, you need to pass X11 a GdkBitmap instance in addition to your pixel buffer. If your RGB buffer has an alpha channel, gdk-pixbuf can take care of the bitmask creation itself, behind the scenes, using the contents of the alpha channel as a guide. Sometimes, however, you might want to create the bitmap yourself, from scratch; perhaps your pixbuf doesn't have an alpha channel, or you need to do some special postprocessing to the mask before you render it to a drawable. If you leave the bitmask creation up to gdk-pixbuf, you won't be able to step in and alter it before gdk-pixbuf passes it to X11.

The gdk_pixbuf_render_threshold_alpha( ) function converts a portion of the pixel buffer's alpha channel into a GdkBitmap mask provided by you (see Figure 10.8 for a visual representation of this function):

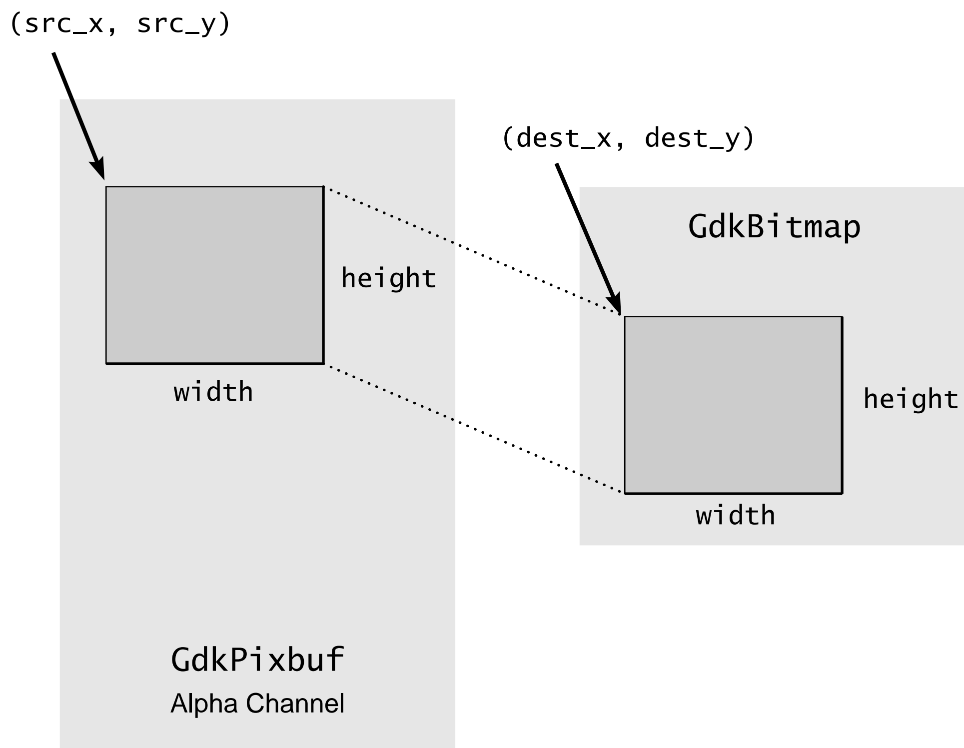


Figure 10.8 Rendering an Alpha Channel to a Bitmap

```
void gdk_pixbuf_render_threshold_alpha(GdkPixbuf *pixbuf,
    GdkBitmap *bitmap, int src_x, int src_y,
    int dest_x, int dest_y, int width, int height,
    int alpha_threshold);
```

The `src_x` and `src_y` parameters determine where to start in the source `GdkPixbuf`; `width` and `height` determine the size of the area to transfer. The `dest_x` and `dest_y` parameters describe where in the destination `GdkBitmap` to render the chosen region. If the `src` coordinates do not match the `dest` coordinates, the bitmask will be offset within `GdkBitmap`. The `GdkBitmap` mask can be a different size from the original pixel buffer, but it must be large enough to contain the complete mask, including any offsets.

With a little juggling of coordinates, you can take any rectangular area from the pixel buffer and translate it into a 1-bit mask of the same size, and then store it anywhere within `GdkBitmap`. However, make sure that the bitmap you pass in is at least (`dest_x` + `width`) pixels wide and (`dest_y` + `height`) pixels tall.

The `alpha_threshold` parameter gives you the critical control over which pixels to turn on in the bitmask and which to turn off. As gdk-pixbuf scans through pixels in your alpha channel, it compares each value to `alpha_threshold`. If the alpha channel value is greater than or equal to `alpha_threshold`, gdk-pixbuf turns on (sets to 1) the corresponding bit in the `GdkBitmap`; any values less than `alpha_threshold` are turned off (set to 0). When the time comes to render an image with that bitmask, X11 copies only pixels that are masked to a 1 and ignores the rest. In a manner of speaking, all areas of the alpha channel that meet or exceed the threshold are rendered. The higher you set the `alpha_threshold`, the less of the image will be rendered.

When your pixel buffer does not contain an alpha channel, `gdk_pixbuf_render_threshold_alpha()` pretends that the entire alpha channel is set to 254. Thus if the `alpha_threshold` parameter is 0 to 254, the mask will consist of only 1's, since the alpha channel will be equal to or above the threshold; everything under that mask will be visible. If you set `alpha_threshold` to 255, the mask will be all 0's, which will hide the entire masked area from view.

When you are done creating your bitmask—which could conceivably amount to multiple calls to `gdk_pixbuf_render_threshold_alpha()`—you should register it with the GDK graphics context as a clipping mask, using the `gdk_gc_set_clip_mask()` function. Later, when GdkRGB renders the image to the target drawable, it will pull your mask from the `GdkGC` structure and use it as a stencil for the image.

Now that you have a GdkPixbuf structure and a 1-bit mask, you are ready to transfer the image to a drawable. Gdk-pixbuf has two functions for rendering to drawables. The first one, gdk_pixbuf_render_to_drawable(), requires that you place the bitmask into a GdkGC structure beforehand, as already described. The second rendering function, gdk_pixbuf_render_to_drawable_alpha(), creates the bitmask from the alpha channel in the pixel buffer, internally using gdk_pixbuf_render_threshold_alpha():

```
void gdk_pixbuf_render_to_drawable(GdkPixbuf *pixbuf,
    GdkDrawable *drawable, GdkGC *gc, int src_x, int src_y,
    int dest_x, int dest_y, int width, int height,
    GdkRgbDither dither, int x_dither, int y_dither);
void gdk_pixbuf_render_to_drawable_alpha(GdkPixbuf *pixbuf,
    GdkDrawable *drawable, int src_x, int src_y,
    int dest_x, int dest_y, int width, int height,
    GdkPixbufAlphaMode alpha_mode, int alpha_threshold,
    GdkRgbDither dither, int x_dither, int y_dither);
```

The former function gives you more flexibility at the risk of greater complexity. You must know how to set up your bitmask and how to set up a GdkGC structure. The second function is easier to use but requires a four-channel RGBA buffer.

For completeness, gdk-pixbuf can transfer images both ways. Not only can it push a pixel buffer out to a drawable, but it can also pull a drawable image into a pixel buffer. This reciprocal capability is perfect for making screen shots or managing special effects such as zooming in on another window. The parameters are fairly straightforward, except for cmap, which is needed only when the drawable is a GdkPixmap instance, since unlike GdkWindow structures, pixmaps don't carry their own color maps:

```
GdkPixbuf *gdk_pixbuf_get_from_drawable(GdkPixbuf *dest,
    GdkDrawable *src, GdkColormap *cmap, int src_x, int src_y,
    int dest_x, int dest_y, int width, int height);
```

10.5.3 Scaling

If you're lucky, the images you load from graphics files will be exactly the correct size for what you need. In the real world, however, you will eventually have to shrink or enlarge some of your images to a more manageable size. Perhaps you need a thumbnail of a graphics file or want to stretch a background to

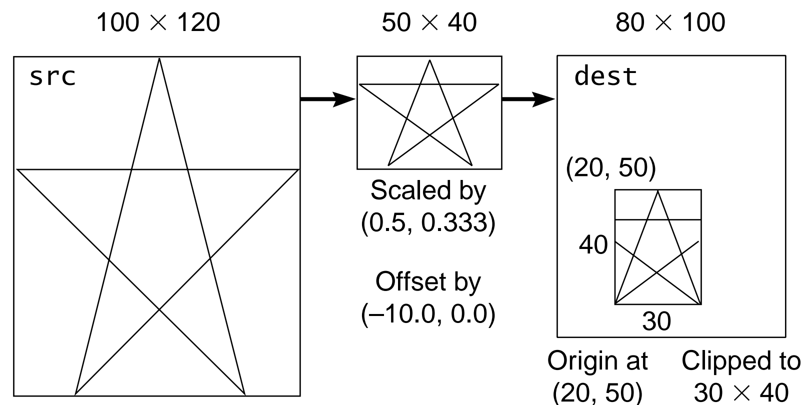
fill a window. Or maybe you just want to display a collection of images all at the same size and need to tweak a few of them to fit. Whatever the reason, image scaling is not something you'll want to tackle from scratch, unless you are very well versed in graphics programming. A poorly designed algorithm is guaranteed to produce noticeable scaling artifacts. Image scaling is a highly mathematical, deeply researched field of study, and we are fortunate to have the algorithms for it already coded and easily accessible in gdk-pixbuf.

The primary scaling function in gdk-pixbuf is `gdk_pixbuf_scale()`, an extremely flexible function that should cover most, if not all, of your scaling needs. It has quite a few parameters and can be confusing at first to use:

```
void gdk_pixbuf_scale(const GdkPixbuf *src, GdkPixbuf *dest,
    int dest_x, int dest_y, int dest_width, int dest_height,
    double offset_x, double offset_y, double scale_x,
    double scale_y, GdkInterpType interp_type);
```

See Figure 10.9 for a visual representation of what the parameters control.

The `src` pixbuf is the base unscaled image—for example, the graphics file you just loaded into memory. The `dest` pixbuf is the preexisting target for rendering the scaled image. The `scale_x` and `scale_y` parameters are the scaling multipliers; a value less than 1.0 will shrink the image, and a value greater than 1.0 will enlarge it. A `(scale_x, scale_y)` value of (2.0, 0.5) will make the




```
gdk_pixbuf_scale(src, dest, 20, 50, 30, 40, -10.0, 0.0, 0.5, 0.333, ART_FILTER_TILES);
```

Figure 10.9 Scaling Parameters

src pixel buffer twice as wide and only half as tall. Gdk-pixbuf will scale the entire src pixbuf by these multipliers. You can then use offset_x and offset_y to move the scaled image in relation to the destination origin at (dest_x, dest_y).

In its final step, gdk_pixbuf_scale( ) copies the area defined by the four dest_* parameters to the dest pixbuf. In other words, gdk_pixbuf_scale( ) scales src by (scale_x, scale_y), moves it to (offset_x, offset_y), and takes a snapshot of it using the rectangle defined by (dest_x, dest_y, dest_width, dest_height). By cleverly manipulating the size and position parameters, you can selectively copy any portion of the scaled src pixbuf. In Figure 10.9 we pass in an offset_x of –10.0 to shift the image 10 pixels to the left so that it's centered in the 30-pixel-wide clipping region in the destination pixbuf.

You must create the dest pixbuf beforehand. You should be careful to create one large enough to hold the scaled image. Always make dest at least (dest_x + dest_width) pixels wide and (dest_y + dest_height) pixels tall.

The last parameter, interp_type, determines which scaling algorithm you want to use. Each of the four algorithms varies in quality, speed, and appearance, so you should pick the filtering style that best suits your needs. GdkInterpType is an enumeration of the available algorithms:

```
typedef enum
{
  GDK_INTERP_NEAREST,
  GDK_INTERP_TILES,
  GDK_INTERP_BILINEAR,
  GDK_INTERP_HYPER
} GdkInterpType;
```

The enumeration represents the order of increasing quality (and increased rendering time) of the filter operations. GDK_INTERP_NEAREST is by far the quickest, but it ends up rather pixelated for enlargements. GDK_INTERP_BILINEAR uses bilinear interpolation to produce smooth, antialiased scaled images, even when you magnify the image to many times its original size. Between NEAREST and BILINEAR is GDK_INTERP_TILES, which looks pixelated like NEAREST when you enlarge but is smoother when you shrink the image, like BILINEAR. GDK_INTERP_HYPER is the highest-quality filter, and of course the slowest. In most cases, TILES and BILINEAR will be good enough.

Often you won't need to use all the various parameters in gdk_pixbuf_scale( ), and they will just get in the way. Fortunately, gdk-pix-

buf also has a streamlined version: gdk_pixbuf_scale_simple(). Gone are the offsets and scaling factors, which are either set to 0 or calculated from the size of the src pixbuf. This function will scale the entire src image to the new size of dest_width × dest_height and create a GdkPixbuf instance containing the scaled image, saving you from creating a properly sized dest pixbuf yourself:

```
GdkPixbuf *gdk_pixbuf_scale_simple(const GdkPixbuf *src,
    int dest_width, int dest_height,
    GdkInterpType interp_type);
```

The two scaling functions we've covered so far will erase the previous contents of the destination pixel buffer (although technically the internally created destination pixel buffer in gdk_pixbuf_scale_simple() will always be initially empty, making this a moot point). Sometimes you'll need to overlay the scaled image on top of an existing image without entirely overwriting it, a process known as **compositing.** The gdk-pixbuf function to do alpha blending like this is gdk_pixbuf_composite(). It is similar to gdk_pixbuf_scale() except for the extra parameter, overall_alpha, which it uses to determine the opacity of the scaled image:

```
void gdk_pixbuf_composite(const GdkPixbuf *src, GdkPixbuf *dest,
    int dest_x, int dest_y, int dest_width, int dest_height,
    double offset_x, double offset_y, double scale_x,
    double scale_y, GdkInterpType interp_type,
    int overall_alpha);
```

The dest pixbuf should already have image data in it; otherwise, what's the point of compositing the scaled src image on top of it?

Incidentally, gdk-pixbuf has a couple of other more specialized scaling and compositing functions: gdk_pixbuf_composite_color() and the slimmer version, gdk_pixbuf_composite_color_simple(). Rather than compositing the scaled image onto another image, these functions will composite it onto a generated checkerboard pattern. This functionality is probably not useful to you unless you are writing an application for viewing or editing anti-aliased images. The checkerboard pattern helps emphasize the anti-aliasing. Figure 10.10 shows testpixbuf-scale, one of the test programs that comes with gdk-pixbuf, using gdk_pixbuf_composite_color() to display gnome-globe.png, an icon packaged with gnome-libs.

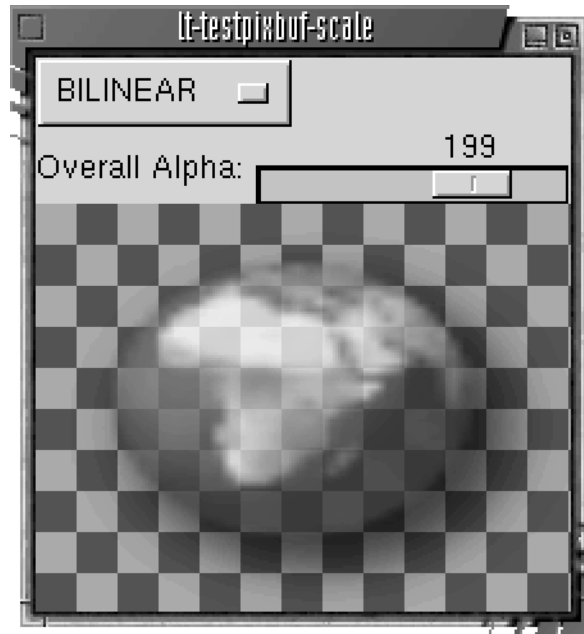


Figure 10.10 Image Composition with `testpixbuf-scale`

A specialization of the scaling operation is the copy operation, in which the scaling multipliers are both 1.0. Rather than forcing you to call `gdk_pixbuf_scale( )` with a one-to-one scaling ratio, gdk-pixbuf offers a convenience function to do it in a more rational way: `gdk_pixbuf_copy_area( )`. This function copies the region defined by `src_x`, `src_y`, `width`, and `height` in the source pixel buffer, into the destination pixel buffer at (`dest_x`, `dest_y`):

```
void gdk_pixbuf_copy_area(const GdkPixbuf *src_pixbuf,
    int src_x, int src_y, int width, int height,
    GdkPixbuf *dest_pixbuf, int dest_x, int dest_y);
```

10.5.4 Progressive Loading

Gdk-pixbuf's image-loading facilities make it very easy for you to load a variety of graphics file formats without doing any extra work. All you have to do is supply the file name, and gdk-pixbuf will handle the rest. However, the file-loading system has one potentially limiting flaw: It blocks until it's finished. With icons and smaller images, this temporary stall is not noticeable enough to matter, but when the file is particularly large, or is loading from a remote site with a slow connection, your application may flounder in an input/output

loop, unable to update itself. The resulting slowness gives your application a buggy, unresponsive appearance, even though it has a good reason for stalling.

To address this problem, the gdk-pixbuf library offers the `GdkPixbufLoader` API. `GdkPixbufLoader` is a derivative of `GtkObject` that allows the loading process to be controlled by your application rather than by gdk-pixbuf's image-loading code. Because it's derived from `GtkObject`, `GdkPixbufLoader` can notify you through GTK+ signals when something interesting happens.

The interface is simple and straightforward. You create a loader object, write chunks of raw image data to it, and then close it when you're done. As soon as the loader receives enough of the graphics file to determine what sort of image it's loading, it creates a `GdkPixbuf` structure in which to store the image data. When you close the loader, it unreferences that `GdkPixbuf`, so if you want the pixbuf still to exist after you're done with the loader, your application will need to explicitly call `gdk_pixbuf_ref()` on it (and of course release it when you're done). The best place to reference the pixbuf is inside an `area_prepared` signal callback, which we'll discuss in a moment.

The loader uses the same code to load images into `GdkPixbufLoader` that gdk-pixbuf uses inside `gdk_pixbuf_new_from_file()`. Here's the basic `GdkPixbufLoader` API:

```
GdkPixbufLoader* gdk_pixbuf_loader_new();
gboolean gdk_pixbuf_loader_write(GdkPixbufLoader *loader,
    const guchar *buf, size_t count);
GdkPixbuf* gdk_pixbuf_loader_get_pixbuf(
    GdkPixbufLoader *loader);
void gdk_pixbuf_loader_close(GdkPixbufLoader *loader);
```

The loader has three signals of interest to us here.[3] Here are the prototypes for these signals, from the `GdkPixbufLoaderClass` structure:

```
void (* area_prepared) (GdkPixbufLoader *loader);
void (* area_updated) (GdkPixbufLoader *loader,
    guint x, guint y, guint width, guint height);
void (* closed) (GdkPixbufLoader *loader);
```

The aforementioned `area_prepared` signal indicates that the loader has read enough of the file to calculate the size and color depth of the image.

3. Actually, `GdkPixbufLoader` has two additional signals, `frame_done` and `animation_done`, which apply to gdk-pixbuf's animation-loading code, but we don't have room to go into that here.

Until the loader has this information, it makes no sense to create the target GdkPixbuf. This first milestone is important enough to merit a special signal emission because it means that your application can allocate space in the display for the image. If you connect to the area_prepared signal, you should take the opportunity in your callback to call gdk_pixbuf_ref( ) on the supplied pixbuf. Also, you are guaranteed at this point that the loader has not started writing into the new pixel buffer, so you can fill it with whatever background you want, whether it's a solid color or a placeholder image. As image data continues to pour into the loader, your background image will slowly be written over, line by line, until the full image arrives.

You'll receive only one area_prepared signal per loader, but altogether you'll end up receiving several area_updated signals. In fact, each time you call gdk_pixbuf_loader_write( ), the loader will fire off another area_updated signal. If you're displaying the progressive image to a drawable—which, after all, is largely the *point* of using GdkPixbufLoader—you should use this opportunity to update the display. In the sample application at the end of this chapter, we do this by copying the currently loaded portion of the image to a GdkPixmap drawable with gdk_pixbuf_render_to_drawable_alpha( ) and then triggering a refresh with gtk_widget_draw( ).

The final signal, closed, is useful if you want to know when the image is completely loaded. If other parts of your application, such as a Web browser window, need to be notified when the image is complete, you can connect them to the loader's closed signal, rather than blocking until then.

10.5.5 Autoconf Support

To facilitate its integration with the GNOME build system, the gdk-pixbuf package contains an autoconf macro, a gdk-pixbuf-config tool, and a drop-in extension for gnome-config.

You can use the macro AM_PATH_GDK_PIXBUF in your configure.in file to run a series of checks for gdk-pixbuf. It works just as you would expect: You give it the minimum version of gdk-pixbuf your application requires, followed by an optional shell-scripted action to perform if the target system has a valid gdk-pixbuf installation, and a second action to perform if the target system fails the test:

```
AM_PATH_GDK_PIXBUF([MINIMUM-VERSION, [ACTION-IF-FOUND
  [, ACTION-IF-NOT-FOUND]]])
```

The macro will call AC_SUBST on the GDK_PIXBUF_CFLAGS and GDK_PIXBUF_LIBS variables, which you can then reference from your

`Makefile.am` file. It also runs some safety checks to alert the administrator of any strange behavior in the current gdk-pixbuf installation. Finally, it adds the options `--with-gdk-pixbuf-prefix` and `--with-gdk-pixbuf-exec-prefix` to the `configure` script to handle cases in which gdk-pixbuf is installed in a nonstandard place on the target system.

Gdk-pixbuf supplies its own version of `gnome-config`, called `gdk-pixbuf-config`, for retrieving path and version information at the command line. It also installs a plug-in extension so that you can get the same information from `gnome-config`. You can run either one of these commands to obtain the necessary `CFLAGS` parameters for gdk-pixbuf:

```
gnome-config gdk_pixbuf --cflags
gdk-pixbuf-config --cflags
```

10.5.6 Gdk-pixbuf Application Example

In our sample graphics application, we will touch on many points related to what we've just been discussing. We will see how to create a `GdkPixbuf` structure from a file and from an existing raw buffer. We will see how to copy image data from buffer to buffer, and from buffer to GDK drawable. We'll dip into image scaling and compositing, event handling, and even progressive image loading. To top things off, we'll include double buffering with a `GdkPixmap`.

Although this example is technically a GNOME application, most of it consists of calls to GDK and gdk-pixbuf. You can click on two of the icons (see Figure 10.11). Clicking on the feather will change the alpha blending level and make the feather fade in and out of view; clicking on the icon for `gnome-error.png` will load an image of a globe, a few scan lines at a time. Figure 10.11 shows a screen shot of the sample application. Listings 10.1 and 10.2 contain the makefile and the source code.

Listing 10.1 Makefile for Sample Gdk-pixbuf Application

```
CC = gcc
CFLAGS = `gnome-config --cflags gnomeui gdk_pixbuf`
LDFLAGS = `gnome-config --libs gnomeui gdk_pixbuf`

rgbtest: rgbtest.c
	$(CC) rgbtest.c -o rgbtest $(CFLAGS) $(LDFLAGS)
```

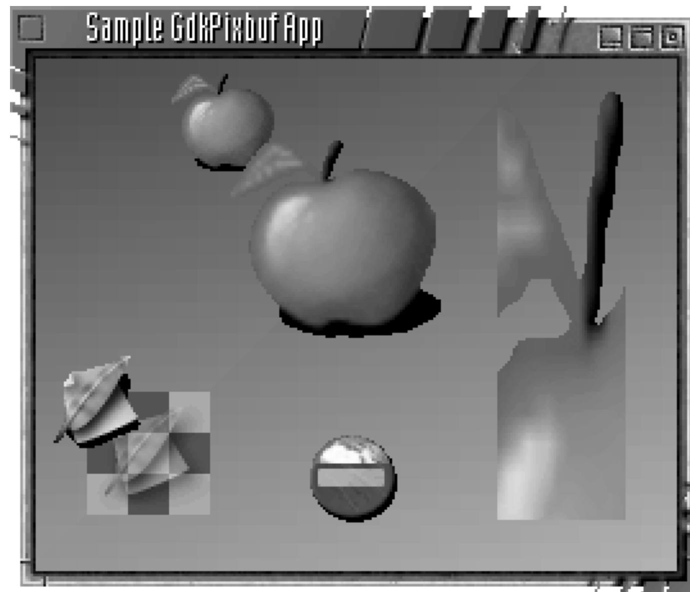


Figure 10.11 Screen Shot of Sample Gdk-pixbuf Application

Listing 10.2 Source Code for Sample Gdk-pixbuf Application

```
/* rgbtest.c - Sample gdk-pixbuf Application */

#include <gnome.h>
#include <gdk-pixbuf/gdk-pixbuf.h>
#include <gdk-pixbuf/gdk-pixbuf-loader.h>

#include <unistd.h>
#include <sys/stat.h>

#define PACKAGE "rgbtest"
#define VERSION "0.0.1"

/* Size of drawing area */
#define WIDTH 250
#define HEIGHT 200

/* Coordinates of progressive loader image */
```

```
#define LOADER_X 100
#define LOADER_Y 140

/* Visible drawing surface */
GtkWidget *drawing_area;

/* Raw pixel buffer */
static guchar drawbuf[WIDTH * HEIGHT * 6];

/* Double-buffer pixmap */
GdkPixmap *dbuf_pixmap = NULL;

/* Feather opacity */
static gint opacity = 127;

/* Loader stuff */
GdkPixbufLoader *loader = NULL;
GdkPixbuf *loader_pixbuf = NULL;
guchar *filebuf = NULL;
guint filesize = 0;
guint curpos = 0;

/* Create a colorful background gradient */
static void init_drawing_buffer( )
{
  gint x, y;
  gint pixel_offset;
  gint rowstride = WIDTH * 3;

  for (y = 0; y < HEIGHT; y++)
  {
    for (x = 0; x < WIDTH; x++)
    {
      pixel_offset = y * rowstride + x * 3;

      drawbuf[pixel_offset] = y * 255 / HEIGHT;
      drawbuf[pixel_offset + 1] = 128 - (x + y) * 255 /
        (WIDTH * HEIGHT);
```

```
      drawbuf[pixel_offset + 2] = x * 255 / WIDTH;
    }
  }
}

/* Render our images to the double-buffered pixmap */
static void render_apples( )
{
  gint width;
  gint height;
  GdkPixbuf *pixbuf1, *pixbuf2, *pixbuf3;

  gchar *applefile = gnome_pixmap_file ("apple-red.png");

  if(applefile)
  {
    g_message ("Rendering apple: %s", applefile);
    pixbuf1 = gdk_pixbuf_new_from_file(applefile);
    width = gdk_pixbuf_get_width(pixbuf1);
    height = gdk_pixbuf_get_height(pixbuf1);

    /* Scale second apple to double size */
    pixbuf2 = gdk_pixbuf_scale_simple(pixbuf1, width * 2,
      height * 2, GDK_INTERP_BILINEAR);

    /* Create a 50 x 180 GdkPixbuf to display our distorted
     * apple. Set the dest parameters to copy into the full
     * area of pixbuf3. Offset the chunk of pixbuf1 that we're
     * scaling by (-50.0, -60.0) so that we see an interesting
     * bit of the stem, and not just the upper left-hand corner
     * of the apple. Finally, stretch the apple by 4 in the
     * horizontal and 12 in the vertical.
     */
    pixbuf3 = gdk_pixbuf_new (GDK_COLORSPACE_RGB, TRUE, 8,
      50, 180);
    gdk_pixbuf_scale(pixbuf1, pixbuf3,
      0, 0, 50, 180,
      -50.0, -60.0,
```

```
      4.0, 12.0,
      GDK_INTERP_BILINEAR);

    gdk_pixbuf_render_to_drawable_alpha(pixbuf1, dbuf_pixmap,
      0, 0, 50, 0, width, height,
      GDK_PIXBUF_ALPHA_BILEVEL, 128,
      GDK_RGB_DITHER_NORMAL, 0, 0);
    gdk_pixbuf_render_to_drawable_alpha(pixbuf2, dbuf_pixmap,
      0, 0, 70, 20, width * 2, height * 2,
      GDK_PIXBUF_ALPHA_BILEVEL, 128,
      GDK_RGB_DITHER_NORMAL, 0, 0);
    gdk_pixbuf_render_to_drawable_alpha(pixbuf3, dbuf_pixmap,
      0, 0, WIDTH - 70, 0, 50, 180,
      GDK_PIXBUF_ALPHA_BILEVEL, 128,
      GDK_RGB_DITHER_NORMAL, 0, 0);

    gdk_pixbuf_unref(pixbuf1);
    gdk_pixbuf_unref(pixbuf2);
    gdk_pixbuf_unref(pixbuf3);
    g_free(applefile);
  }
}

static void render_feathers( )
{
  gint width;
  gint height;
  GdkPixbuf *pixbuf1, *pixbuf2;

  gchar *featherfile = gnome_pixmap_file ("gnome-word.png");

  if(featherfile)
  {
    g_message ("Rendering feather: %s", featherfile);
    pixbuf1 = gdk_pixbuf_new_from_file(featherfile);
    width = gdk_pixbuf_get_width(pixbuf1);
    height = gdk_pixbuf_get_height(pixbuf1);
```

```
    /* Create a feather composited onto a checkerboard */
    pixbuf2 = gdk_pixbuf_composite_color_simple(pixbuf1,
      width, height, GDK_INTERP_TILES, opacity, 16,
      0xaaaaaa, 0x555555);

    /* Render checkerboard feather first, so the overlapping
     * normal feather (pixbuf1) will come out on top
     */
    gdk_pixbuf_render_to_drawable_alpha(pixbuf2, dbuf_pixmap,
      0, 0, 20, 130,
      width * 1, height * 1,
      GDK_PIXBUF_ALPHA_BILEVEL, 128,
      GDK_RGB_DITHER_NORMAL, 0, 0);
    gdk_pixbuf_render_to_drawable_alpha(pixbuf1, dbuf_pixmap,
      0, 0, 0, 110,
      width, height,
      GDK_PIXBUF_ALPHA_BILEVEL, 128,
      GDK_RGB_DITHER_NORMAL, 0, 0);

    gdk_pixbuf_unref(pixbuf1);
    gdk_pixbuf_unref(pixbuf2);
    g_free(featherfile);
  }
}

void expose_event_cb(GtkWidget *widget, GdkEventExpose *event,
    gpointer data)
{
  /* Don't repaint entire window upon each exposure */
  gdk_window_set_back_pixmap (widget->window, NULL, FALSE);

  /* Refresh double buffer, then copy the "dirtied" area to
   * the on-screen GdkWindow
   */
  gdk_window_copy_area(widget->window,
    widget->style->fg_gc[GTK_STATE_NORMAL],
    event->area.x, event->area.y,
    dbuf_pixmap,
```

```
    event->area.x, event->area.y,
    event->area.width, event->area.height);
}

static void area_prepared_cb(GdkPixbufLoader *loader,
    gint x, gint y, gint width, gint height, gpointer data)
{
  gchar *bkgd_file;
  GdkPixbuf *pixbuf;

  loader_pixbuf = gdk_pixbuf_loader_get_pixbuf(loader);
  gdk_pixbuf_ref(loader_pixbuf);

  bkgd_file = gnome_pixmap_file ("gnome-error.png");
  pixbuf = gdk_pixbuf_new_from_file(bkgd_file);
  g_free(bkgd_file);

  /* Copy placeholder image to loader_pixbuf */
  gdk_pixbuf_copy_area(pixbuf, 0, 0,
    gdk_pixbuf_get_width(pixbuf),
    gdk_pixbuf_get_height(pixbuf),
    loader_pixbuf,
    0, 0);
}

static void area_updated_cb(GdkPixbufLoader *loader,
    gpointer data)
{
  gint height, width;
  GdkRectangle rect = { LOADER_X, LOADER_Y, 48, 48 };

  g_message("Rendering Loader Pixbuf");
  width = gdk_pixbuf_get_width(loader_pixbuf);
  height = gdk_pixbuf_get_height(loader_pixbuf);

  /* Copy latest version of progressive image to the double
   * buffer
   */
```

```
  gdk_pixbuf_render_to_drawable_alpha(loader_pixbuf,
    dbuf_pixmap,
    0, 0, LOADER_X, LOADER_Y,
    width, height,
    GDK_PIXBUF_ALPHA_BILEVEL, 128,
    GDK_RGB_DITHER_NORMAL, 0, 0);

  /* Trigger an expose_event to flush the changes to the
   * drawing area
   */
  gtk_widget_draw(drawing_area, &rect);
}

static void nudge_loader( )
{
  guint bitesize = 256;
  guint writesize = bitesize;

  if (curpos >= filesize)
    return;

  /* Trim writesize if it extends past the end of the file */
  if (curpos + bitesize >= filesize)
    writesize = filesize - curpos;

  /* Send next chunk of image */
  if (!gdk_pixbuf_loader_write(loader, &filebuf[curpos],
      writesize))
    g_warning("Loader write failed!!");

  curpos += writesize;

  /* Clean up loader when we're finished writing the entire
   * file; loader_pixbuf is still around because we referenced it
   */
  if (curpos >= filesize)
    gdk_pixbuf_loader_close(loader);
}
```

```
static void init_loader()
{
  FILE *file;
  gchar *loadfile;

  loader = gdk_pixbuf_loader_new();

  gtk_signal_connect(GTK_OBJECT(loader), "area_prepared",
    GTK_SIGNAL_FUNC(area_prepared_cb), NULL);
  gtk_signal_connect(GTK_OBJECT(loader), "area_updated",
    GTK_SIGNAL_FUNC(area_updated_cb), NULL);

  /* Load file into memory for easier cycling */
  loadfile = gnome_pixmap_file ("gnome-globe.png");
  file = fopen (loadfile, "r");
  if (file)
  {
    struct stat statbuf;

    if (stat(loadfile, &statbuf) == 0)
    {
      filesize = statbuf.st_size;
      g_message("Found file %s, size=%d", loadfile, filesize);

      filebuf = g_malloc(filesize);
      fread(filebuf, sizeof(char), filesize, file);
    }
    fclose(file);
  }
  else
    g_warning("Failed to find file %s", loadfile);

  /* Load the first small chunk, i.e., enough to trigger an
   * area_prepared event
   */
  nudge_loader();
```

```
  g_free(loadfile);
}

gint event_cb(GtkWidget *widget, GdkEvent *event, gpointer data)
{
  GdkEventButton *bevent = (GdkEventButton *)event;

  switch ((gint)event->type)
  {
  case GDK_BUTTON_PRESS:
    /* Handle mouse button press */

    /* For clicks on the feather icon */
    if (bevent->x >= 20 && bevent->x <= 68 &&
      bevent->y >= 130 && bevent->y <= 178)
    {
      GdkRectangle rect = { 20, 130, 48, 48 };

      opacity += 63;
      if (opacity > 255)
        opacity = 0;

      render_feathers( );
      gtk_widget_draw(widget, &rect);
    }

    /* For clicks on the globe loader icon */
    if (bevent->x >= 100 && bevent->x <= 148 &&
      bevent->y >= 140 && bevent->y <= 188)
    {
      nudge_loader( );
    }
    return TRUE;
  }

  /* Event not handled; try parent item */
  return FALSE;
}
```

```
void quit_cb (GtkWidget *widget)
{
  gdk_pixbuf_unref(loader_pixbuf);
  g_free(filebuf);
  gtk_main_quit ();
}

int main (int argc, char **argv)
{
  GtkWidget *app;
  GdkPixbuf *pixbuf;

  /* Includes a call to gdk_rgb_init() */
  gnome_init (PACKAGE, VERSION, argc, argv);

  gtk_widget_push_visual(gdk_rgb_get_visual());
  gtk_widget_push_colormap(gdk_rgb_get_cmap());
  app = gnome_app_new(PACKAGE, "Sample GdkPixbuf App");
  gtk_widget_pop_visual();
  gtk_widget_pop_colormap();

  /* Restrict window resizing to size of drawing buffer */
  gtk_widget_set_usize(GTK_WIDGET(app), WIDTH, HEIGHT);
  gtk_window_set_policy(GTK_WINDOW(app), TRUE, FALSE, FALSE);

  /* Handle window manager closing */
  gtk_signal_connect(GTK_OBJECT(app), "delete_event",
    GTK_SIGNAL_FUNC(quit_cb), NULL);

  /* Create drawing-area widget */
  drawing_area = gtk_drawing_area_new ();
  gtk_widget_add_events(GTK_WIDGET(drawing_area),
    GDK_BUTTON_PRESS_MASK);
  gtk_signal_connect(GTK_OBJECT(drawing_area), "expose_event",
    GTK_SIGNAL_FUNC(expose_event_cb), NULL);
  gtk_signal_connect(GTK_OBJECT(drawing_area), "event",
    GTK_SIGNAL_FUNC(event_cb), NULL);
```

```
  gnome_app_set_contents(GNOME_APP(app), drawing_area);

  gtk_widget_show_all(app);

  /* Create double-buffered pixmap; must do it here, after
   * app->window is created. Inherits color map and visual
   * from app.
   */
  dbuf_pixmap = gdk_pixmap_new(app->window, WIDTH, HEIGHT, -1);

  /* Create a GdkPixbuf out of our background gradient, then
   * copy it to our double-buffered pixmap */
  init_drawing_buffer( );
  pixbuf = gdk_pixbuf_new_from_data(drawbuf, GDK_COLORSPACE_RGB,
    FALSE, 8, WIDTH, HEIGHT, WIDTH * 3, NULL, NULL);
  gdk_pixbuf_render_to_drawable(pixbuf, dbuf_pixmap,
    app->style->fg_gc[GTK_STATE_NORMAL],
    0, 0, 0, 0, WIDTH, HEIGHT,
    GDK_RGB_DITHER_NORMAL, 0, 0);

  init_loader( );
  render_apples( );
  render_feathers( );

  gtk_main ( );

  return 0;
}
```

This sample application works by creating a `GdkPixmap` resource to hold our double buffer and a `GtkDrawingArea` widget to display our graphical images. We use instances of `GdkPixbuf` to create and load smaller parts of the main image, which we then copy into our `GdkPixmap` with `gdk_pixbuf_render_to_drawable_alpha( )`. Each time we get an `expose_event` signal, we copy to the drawing-area widget only the portion

that needs refreshing. When we change the double buffer, we must trigger an expose_event signal for that region with a call to gtk_widget_draw().

The action starts in main(). We create a GnomeApp widget for our top-level window, pushing the GdkRGB visual and color map while we do so. This extra step ensures that we have a compatible color setup for gdk-pixbuf. Next we set the resizing policy so that the user can't resize the main window to larger than our drawing area. After that, we connect GTK+ signal handlers for the delete_event signal (to clean up on exit), the event signal (to handle mouse button presses), and the expose_event signal (to copy "dirty" areas from the double buffer to the drawing area). Finally, we create our GdkPixmap double buffer and load it with various icon images from the gnome-core package.

Our expose_event handler is quite simple. All it does is copy a rectangular area from the double buffer into the drawing area. It uses the coordinates passed in with event->area to avoid copying more than it has to, to restore the window after a resize or exposure.

After rendering the background gradient and the apple icons the first time, we don't have to worry about them again for the life of the application. Their images will continue to reside in the double buffer because we never do anything to overwrite them. The only areas we need to update after the initial rendering are the two areas we set up as clickable regions, in the event handler. To minimize the amount of extra work we do and increase the performance of our application, we set up a clipping region for these two areas by passing a GdkRectangle instance into gtk_widget_draw() when we update their images in the double buffer.

The way we handle mouse clicks is a little heavy-handed. We end up hardcoding the coordinates for the two clickable regions. If the user clicks inside the region we define for the feather icon, we increase the opacity and redraw the feather. If the mouse click lands on the globe icon, we update that icon through the progressive loader. We ignore mouse clicks everywhere else. This approach is fairly manageable for a couple of small icons that never move, but it isn't scalable at all. If we had movable objects, we would have to add a lot of extra code to keep track of the clickable "hot spots." This type of code can become complex very quickly, and it probably isn't something you'll want to attempt if you don't have to. The GNOME Canvas (see Chapter 11) addresses exactly these issues and abstracts away a great deal of the complexity of dealing with movable graphics. If you're doing anything more complex than displaying static images, you should consider using the Canvas instead of GtkDrawingArea.

The progressive loader updates the globe icon. Inside our init_loader() function, we set up the callbacks for the loader, load the entire gnome-

globe.png file into filebuf, and then call nudge_loader() once to write the first bit of the file into the loader's pixel buffer. This action loads enough image data to determine the size and color depth of the image, which in turn triggers the area_prepared signal. In the area_prepared callback, we set up a temporary placeholder icon, using the gnome-error.png graphic. We use the gdk_pixbuf_copy_area() function to copy the image from one pixel buffer to another.

Each time you click on this icon, the application calls nudge_loader() again, which writes out another few scan lines, slowly replacing the background gnome-error.png placeholder image with the foreground gnome-globe.png image. When the image is completely loaded, we call gdk_pixbuf_loader_close() to free up the loader. At this point, if we hadn't referenced loader_pixbuf in the area_prepared callback, that pixbuf would have been destroyed along with the loader. As it is, the loader_pixbuf will remain until we unreference it in the delete_event callback.

You may have noticed from the code or from the jagged edges in the screen shot that we are not achieving the best graphics quality possible. Even though the graphics are all four-channel RGBA images, fully capable of alpha blending, we fail to take advantage of the alpha channel because our double buffer is an X11 pixmap, not a GdkPixbuf instance. Because we render each image separately to the server-side pixmap, the alpha channel is reduced to a 1-bit mask for each copy operation. If we had used GdkPixbuf for the double buffer instead of GdkPixmap, we could have kept the full range of the alpha channel intact, leading to smoother edges on the apples and better blending with the background gradient. Consider this a challenge to modify the sample application to use full alpha blending.