Smart_Ch05f.qxd 6/10/05 11:17 AM Page 131

CHAPTER 5

Drawing and Printing

This chapter introduces the idea of the *device context*, generalizing the concept of a drawing surface such as a window or a printed page. We will discuss the available device context classes and the set of "drawing tools" that wxWidgets provides for handling fonts, color, line drawing, and filling. Next we describe a device context's drawing functions and how to use the wxWidgets printing framework. We end the chapter by briefly discussing `wxGLCanvas`, which provides a way for you to draw 3D graphics on your windows using OpenGL.

UNDERSTANDING DEVICE CONTEXTS

All drawing in wxWidgets is done on a device context, using an instance of a class derived from `wxDC`. There is no such thing as drawing directly to a window; instead, you create a device context for the window and then draw on the device context. There are also device context classes that work with bitmaps and printers, or you can design your own. A happy consequence of this abstraction is that you can define drawing code that will work on a number of different device contexts: just parameterize it with `wxDC`, and if necessary, take into account the device's resolution by scaling appropriately. Let's describe the major properties of a device context.

A device context has a coordinate system with its origin at the top-left of the surface. This position can be changed with `SetDeviceOrigin` so that graphics subsequently drawn on the device context are shifted—this is used when painting with `wxScrolledWindow`. You can also use `SetAxisOrientation` if you prefer, say, the *y*-axis to go from bottom to top.

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 132

There is a distinction between *logical units* and *device units*. Device units are the units native to the particular device—for a screen, a device unit is a pixel. For a printer, the device unit is defined by the resolution of the printer, which can be queried using `GetSize` (for a page size in device units) or `GetSizeMM` (for a page size in millimeters).

The *mapping mode* of the device context defines the unit of measurement used to convert logical units to device units. Note that some device contexts, in particular `wxPostScriptDC`, do not support mapping modes other than `wxMM_TEXT`. Table 5-1 lists the available mapping modes.

Table 5-1 Mapping Modes

`wxMM_TWIPS`	Each logical unit is 1/20 of a point, or 1/1440 of an inch.
`wxMM_POINTS`	Each logical unit is a point, or 1/72 of an inch.
`wxMM_METRIC`	Each logical unit is 1 millimeter.
`wxMM_LOMETRIC`	Each logical unit is 1/10 of a millimeter.
`wxMM_TEXT`	Each logical unit is 1 pixel. This is the default mode.

You can impose a further scale on your logical units by calling `SetUser Scale`, which multiplies with the scale implied by the mapping mode. For example, in `wxMM_TEXT` mode, a user scale value of (1.0, 1.0) makes logical and device units identical. By default, the mapping mode is `wxMM_TEXT`, and the scale is (1.0, 1.0).

A device context has a clipping region, which can be set with `SetClipping Region` and cleared with `DestroyClippingRegion`. Graphics will not be shown outside the clipping region. One use of this is to draw a string so that it appears only inside a particular rectangle, even though the string might extend beyond the rectangle boundary. You can set the clipping region to be the same size and location as the rectangle, draw the text, and then destroy the clipping region, and the text will be truncated to fit inside the rectangle.

Just as with real artistry, in order to draw, you must first select some tools. Any operation that involves drawing an outline uses the currently selected pen, and filled areas use the current brush. The current font, together with the foreground and background text color, determines how text will appear. We will discuss these tools in detail later, but first we'll look at the types of device context that are available to us.

Available Device Contexts

These are the device context classes you can use:

- ☞ `wxClientDC`. For drawing on the client area of a window.
- ☞ `wxBufferedDC`. A replacement for `wxClientDC` for double-buffered painting.

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 133

- ☞ `wxWindowDC`. For drawing on the client and non-client (decorated) area of a window. This is rarely used and not fully implemented on all platforms.
- ☞ `wxPaintDC`. For drawing on the client area of a window during a paint event handler.
- ☞ `wxBufferedPaintDC`. A replacement for `wxPaintDC` for double-buffered painting.
- ☞ `wxScreenDC`. For drawing on or copying from the screen.
- ☞ `wxMemoryDC`. For drawing into or copying from a bitmap.
- ☞ `wxMetafileDC`. For creating a metafile (Windows and Mac OS X).
- ☞ `wxPrinterDC`. For drawing to a printer.
- ☞ `wxPostScriptDC`. For drawing to a PostScript file or printer.

The following sections describe how to create and work with these device contexts. Working with printer device contexts is discussed in more detail later in the chapter in “Using the Printing Framework.”

Drawing on Windows with *wxClientDC*

Use `wxClientDC` objects to draw on the client area of windows outside of paint events. For example, to implement a doodling application, you might create a `wxClientDC` object within your mouse event handler. It can also be used within background erase events.

Here’s a code fragment that demonstrates how to paint on a window using the mouse:

```
BEGIN_EVENT_TABLE(MyWindow, wxWindow)
    EVT_MOTION(MyWindow::OnMotion)
END_EVENT_TABLE()

void MyWindow::OnMotion(wxMouseEvent& event)
{
    if (event.Dragging())
    {
        wxClientDC dc(this);
        wxPen pen(*wxRED, 1); // red pen of width 1
        dc.SetPen(pen);
        dc.DrawPoint(event.GetPosition());
        dc.SetPen(wxNullPen);
    }
}
```

For more realistic doodling code, see Chapter 19, "Working with Documents and Views." The "Doodle" example uses line segments instead of points and implements undo/redo. It also stores the line segments, so that when the window is repainted, the graphic is redrawn; using the previous code, the graphic will only linger on the window until the next paint event is received. You may also want to use `CaptureMouse` and `ReleaseMouse` to direct all mouse events to your window while the mouse button is down.

An alternative to using `wxClientDC` directly is to use `wxBufferedDC`, which stores your drawing in a memory device context and transfers it to the window in one step when the device context is about to be deleted. This can result in smoother updates—for example, if you don't want the user to see a complex graphic being updated bit by bit. Use the class exactly as you would use `wxClientDC`. For efficiency, you can pass a stored bitmap to the constructor to avoid the object re-creating a bitmap each time.

Erasing Window Backgrounds

A window receives two kinds of paint event: `wxPaintEvent` for drawing the main graphic, and `wxEraseEvent` for painting the background. If you just handle `wxPaintEvent`, the default `wxEraseEvent` handler will clear the background to the color previously specified by `wxWindow::SetBackgroundColour`, or a suitable default.

This may seem rather convoluted, but this separation of background and foreground painting enables maximum control on platforms that follow this model, such as Windows. For example, suppose you want to draw a textured background on a window. If you tile your texture bitmap in `OnPaint`, you will see a brief flicker as the background is cleared prior to painting the texture. To avoid this, handle `wxEraseEvent` and do nothing in the handler. Alternatively, you can do the background tiling in the erase handler, and paint the foreground in the paint handler (however, this defeats buffered drawing as described in the next section).

On some platforms, intercepting `wxEraseEvent` still isn't enough to suppress default background clearing. The safest thing to do if you want to have a background other than a plain color is to call `wxWindow::SetBackgroundStyle` passing `wxBG_STYLE_CUSTOM`. This tells wxWidgets to leave all background painting to the application.

If you do decide to implement an erase handler, call `wxEraseEvent::GetDC` and use the returned device context if it exists. If it's `NULL`, you can use a `wxClientDC` instead. This allows for wxWidgets implementations that don't pass a device context to the erase handler, which can be an unnecessary expense if it's not used. This is demonstrated in the following code for drawing a bitmap as a background texture:

```
BEGIN_EVENT_TABLE(MyWindow, wxWindow)
  EVT_ERASE_BACKGROUND(MyWindow::OnErase)
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 135

```
END_EVENT_TABLE()

void MyWindow::OnErase(wxEraseEvent& event)
{
    wxClientDC* clientDC = NULL;
    if (!event.GetDC())
        clientDC = new wxClientDC(this);

    wxDC* dc = clientDC ? clientDC : event.GetDC() ;

    wxSize sz = GetClientSize();
    wxEffects effects;
    effects.TileBitmap(wxRect(0, 0, sz.x, sz.y), *dc, m_bitmap);

    if (clientDC)
        delete clientDC;
}
```

As with paint events, the device context will be clipped to the area that needs to be repaired, if using the object returned from `wxEraseEvent::GetDC`.

Drawing on Windows with *wxPaintDC*

If you define a paint event handler, you must always create a `wxPaintDC` object, even if you don't use it. Creating this object will tell wxWidgets that the invalid regions in the window have been repainted so that the windowing system won't keep sending paint events *ad infinitum*. In a paint event, you can call `wxWindow::GetUpdateRegion` to get the region that is invalid, or `wxWindow::IsExposed` to determine if the given point or rectangle is in the update region. If possible, just repaint this region. The device context will automatically be clipped to this region anyway during the paint event, but you can speed up redraws by only drawing what is necessary.

Paint events are generated when user interaction causes regions to need repainting, but they can also be generated as a result of `wxWindow::Refresh` or `wxWindow::RefreshRect` calls. If you know exactly which area of the window needs to be repainted, you can invalidate that region and cause as little flicker as possible. One problem with refreshing the window this way is that you can't guarantee exactly when the window will be updated. If you really need to have the paint handler called immediately—for example, if you're doing time-consuming calculations—you can call `wxWindow::Update` after calling `Refresh` or `RefreshRect`.

The following code draws a red rectangle with a black outline in the center of a window, if the rectangle was in the update region:

```
BEGIN_EVENT_TABLE(MyWindow, wxWindow)
  EVT_PAINT(MyWindow::OnPaint)
END_EVENT_TABLE()

void MyWindow::OnPaint(wxPaintEvent& event)
{
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 136

```
    wxPaintDC dc(this);

    dc.SetPen(*wxBLACK_PEN);
    dc.SetBrush(*wxRED_BRUSH);

    // Get window dimensions
    wxSize sz = GetClientSize();

    // Our rectangle dimensions
    wxCoord w = 100, h = 50;

    // Center the rectangle on the window, but never
    // draw at a negative position.
    int x = wxMax(0, (sz.x—w)/2);
    int y = wxMax(0, (sz.y—h)/2);

    wxRect rectToDraw(x, y, w, h);

    // For efficiency, do not draw if not exposed
    if (IsExposed(rectToDraw))
        DrawRectangle(rectToDraw);
}
```

Note that by default, when a window is resized, only the newly exposed areas are included in the update region. Use the `wxFULL_REPAINT_ON_RESIZE` window style to have the entire window included in the update region when the window is resized. In our example, we need this style because resizing the window changes the position of the graphic, and we need to make sure that no odd bits of the rectangle are left behind.

`wxBufferedPaintDC` is a buffered version of `wxPaintDC`. Simply replace `wxPaintDC` with `wxBufferedPaintDC` in your paint event handler, and the graphics will be drawn to a bitmap before being drawn all at once on the window, reducing flicker.

As we mentioned in the previous topic, another thing you can do to make drawing smoother (particularly when resizing) is to paint the background in your paint handler, and not in an erase background handler. All the painting will then be done in your buffered paint handler, so you don't see the background being erased before the paint handler is called. Add an empty erase background handler, and call `SetBackgroundStyle` with `wxBG_STYLE_CUSTOM` to hint to some systems not to clear the background automatically. In a scrolling window, where the device origin is moved to shift the graphic for the current scroll position, you will need to calculate the position of the window client area for the current origin. The following code snippet illustrates how to achieve smooth painting and scrolling for a class derived from `wxScrolledWindow`:

```
#include "wx/dcbuffer.h"

BEGIN_EVENT_TABLE(MyCustomCtrl, wxScrolledWindow)
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 137

```
    EVT_PAINT(MyCustomCtrl::OnPaint)
    EVT_ERASE_BACKGROUND(MyCustomCtrl::OnEraseBackground)
END_EVENT_TABLE()

/// Painting
void MyCustomCtrl::OnPaint(wxPaintEvent& event)
{
    wxBufferedPaintDC dc(this);

    // Shifts the device origin so we don't have to worry
    // about the current scroll position ourselves
    PrepareDC(dc);

    // Paint the background
    PaintBackground(dc);

    // Paint the graphic
    ...
}

/// Paint the background
void MyCustomCtrl::PaintBackground(wxDC& dc)
{
    wxColour backgroundColour = GetBackgroundColour();
    if (!backgroundColour.Ok())
        backgroundColour =
            wxSystemSettings::GetColour(wxSYS_COLOUR_3DFACE);

    dc.SetBrush(wxBrush(backgroundColour));
    dc.SetPen(wxPen(backgroundColour, 1));

    wxRect windowRect(wxPoint(0, 0), GetClientSize());

    // We need to shift the client rectangle to take into account
    // scrolling, converting device to logical coordinates
    CalcUnscrolledPosition(windowRect.x, windowRect.y,
                           & windowRect.x, & windowRect.y);
    dc.DrawRectangle(windowRect);
}

// Empty implementation, to prevent flicker
void MyCustomCtrl::OnEraseBackground(wxEraseEvent& event)
{
}
```

To increase efficiency when using `wxBufferedPaintDC`, you can maintain a bitmap large enough to cope with all window sizes (for example, the screen size) and pass it to the `wxBufferedPaintDC` constructor as its second argument. Then the device context doesn't have to keep creating and destroying its bitmap for every paint event.

The area that `wxBufferedPaintDC` copies from its buffer is normally the size of the window client area (the part that the user can see). The actual paint context that is internally created by the class is *not* transformed by the window's `PrepareDC` to reflect the current scroll position. However, you can specify that both the paint context and your buffered paint context use the same

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 138

transformations by passing `wxBUFFER_VIRTUAL_AREA` to the `wxBufferedPaintDC` constructor, rather than the default `wxBUFFER_CLIENT_AREA`. Your window's `PrepareDC` function will be called on the actual paint context so the transformations on both device contexts match. In this case, you will need to supply a bitmap that is the same size as the virtual area in your scrolled window. This is inefficient and should normally be avoided. Note that at the time of writing, using buffering with `wxBUFFER_CLIENT_AREA` does not work with scaling (`SetUserScale`).

For a full example of using `wxBufferedPaintDC`, you might like to look at the `wxThumbnailCtrl` control in `examples/chap12/thumbnail` on the CD-ROM.

Drawing on Bitmaps with *wxMemoryDC*

A memory device context has a bitmap associated with it, so that drawing into the device context draws on the bitmap. First create a `wxMemoryDC` object with the default constructor, and then use `SelectObject` to associate a bitmap with the device context. When you have finished with the device context, you should call `SelectObject` with `wxNullBitmap` to remove the association.

The following example creates a bitmap and draws a red rectangle outline on it:

```
wxBitmap CreateRedOutlineBitmap()
{
    wxMemoryDC memDC;
    wxBitmap bitmap(200, 200);
    memDC.SelectObject(bitmap);
    memDC.SetBackground(*wxWHITE_BRUSH);
    memDC.Clear();
    memDC.SetPen(*wxRED_PEN);
    memDC.SetBrush(*wxTRANSPARENT_BRUSH);
    memDC.DrawRectangle(wxRect(10, 10, 100, 100));
    memDC.SelectObject(wxNullBitmap);
    return bitmap;
}
```

You can also copy areas from a memory device context to another device context with the `Blit` function, described later in the chapter.

Creating Metafiles with *wxMetafileDC*

`wxMetafileDC` is available on Windows and Mac OS X, where it models a drawing surface for a Windows metafile or a Mac PICT, respectively. It allows you to draw into a `wxMetafile` object, which consists of a list of drawing instructions that can be interpreted by an application or rendered into a device context with `wxMetafile::Play`.

Accessing the Screen with *wxScreenDC*

Use `wxScreenDC` for drawing on areas of the whole screen. This is useful when giving feedback for dragging operations, such as the sash on a splitter window. For efficiency, you can limit the screen area to be drawn on to a specific region (often the dimensions of the window in question). As well as drawing with this class, you can copy areas to other device contexts and use it for capturing screenshots. Because it is not possible to control where other applications are drawing, use of `wxScreenDC` to draw on the screen usually works best when restricted to windows in the current application.

Here's example code that snaps the current screen and returns it in a bitmap:

```
wxBitmap GetScreenShot()
{
    wxSize screenSize = wxGetDisplaySize();
    wxBitmap bitmap(screenSize.x, screenSize.y);
    wxScreenDC dc;
    wxMemoryDC memDC;
    memDC.SelectObject(bitmap);
    memDC.Blit(0, 0, screenSize.x, screenSize.y, & dc, 0, 0);
    memDC.SelectObject(wxNullBitmap);
    return bitmap;
}
```

Printing with *wxPrinterDC* and *wxPostScriptDC*

`wxPrinterDC` represents the printing surface. On Windows and Mac, it maps to the respective printing system for the application. On other Unix-based systems where there is no standard printing model, a `wxPostScriptDC` is used instead, unless GNOME printing support is available (see the later section, "Printing Under Unix with GTK+").

There are several ways to construct a `wxPrinterDC` object. You can pass it a `wxPrintData` after setting paper type, landscape or portrait, number of copies, and so on. An easier way is to show a `wxPrintDialog` and then call `wxPrintDialog::GetPrintDC` to retrieve an appropriate `wxPrinterDC` for the settings chosen by the user. At a higher level, you can derive a class from `wxPrintout` to specify behavior for printing and print previewing, passing it to an instance of `wxPrinter` (see the later section on printing).

If your printout is mainly text, consider using the `wxHtmlEasyPrinting` class to bypass the need to deal with `wxPrinterDC` or `wxPrintout` altogether: just write an HTML file (using wxWidgets' subset of HTML syntax) and create a `wxHtmlEasyPrinting` object to print or preview it. This could save you days or even weeks of programming to format your text, tables, and images. See Chapter 12, "Advanced Window Classes," for more on this.

`wxPostScriptDC` is a device context specifically for creating PostScript files for sending to a printer. Although mainly for Unix-based systems, it can be

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 140

used on other systems too, where PostScript files need to be generated and you can't guarantee the presence of a PostScript printer driver.

You can create a `wxPostScriptDC` either by passing a `wxPrintData` object, or by passing a file name, a boolean to specify whether a print dialog should be shown, and a parent window for the dialog. For example:

```
#include "wx/dcps.h"

wxPostScriptDC dc(wxT("output.ps"), true, wxGetApp().GetTopWindow());

if (dc.Ok())
{
    // Tell it where to find the AFM files
    dc.GetPrintData().SetFontMetricPath(wxGetApp().GetFontPath());

    // Set the resolution in points per inch (the default is 720)
    dc.SetResolution(1440);

    // Draw on the device context
    ...
}
```

One of the quirks of `wxPostScriptDC` is that it can't directly return text size information from `GetTextExtent`. You will need to provide AFM (Adobe Font Metric) files with your application and use `wxPrintData::SetFontMetricPath` to specify where wxWidgets will find them, as in this example. You can get a selection of GhostScript AFM files from `ftp://biolpc22.york.ac.uk/pub/support/gs_afm.tar.gz`.

DRAWING TOOLS

Drawing code in wxWidgets operates like a very fast artist, rapidly selecting colors and drawing tools, drawing a little part of the scene, then selecting different tools, drawing another part of the scene, and so on. Here we describe the `wxColour`, `wxPen`, `wxBrush`, `wxFont` and `wxPalette` classes. You will also find it useful to refer to the descriptions of other classes relevant to drawing—`wxRect`, `wxRegion`, `wxPoint`, and `wxSize`, which are described in Chapter 13, "Data Structure Classes."

Note that these classes use "reference-counting," efficiently copying only internal pointers rather than chunks of memory. In most circumstances, you can create color, pen, brush, and font objects on the stack as they are needed without worrying about speed implications. If your application does have performance problems, you can take steps to improve efficiency, such as storing some objects as data members.

wxColour

You use wxColour to define various aspects of color when drawing. (Because wxWidgets started life in Edinburgh, the API uses British spelling. However, to cater for the spelling sensibilities of the New World, wxWidgets defines wxColor as an alias for wxColour.)

You can specify the text foreground and background color for a device context using a device context's SetTextForeground and SetTextBackground functions, and you also use wxColour to create pens and brushes.

A wxColour object can be constructed in a number of different ways. You can pass red, green, and blue values (each 0 to 255), or a standard color string such as WHITE or CYAN, or you can create it from another wxColour object. Alternatively, you can use the stock color objects, which are pointers: wxBLACK, wxWHITE, wxRED, wxBLUE, wxGREEN, wxCYAN, and wxLIGHT_GREY. The stock object wxNullColour is an uninitialized color for which the Ok function always returns false.

Using the wxSystemSettings class, you can retrieve some standard, system-wide colors, such as the standard 3D surface color, the default window background color, menu text color, and so on. Please refer to the documentation for wxSystemSettings::GetColour for the identifiers you can pass.

Here are some different ways to create a wxColour object:

```
wxColour color(0, 255, 0); // green
wxColour color(wxT("RED")); // red

// The color used for 3D faces and panels
wxColour color(wxSystemSettings::GetColour(wxSYS_COLOUR_3DFACE));
```

You can also use the wxTheColourDatabase pointer to add a new color, find a wxColour object for a given name, or find the name corresponding to the given color:

```
wxTheColourDatabase->Add(wxT("PINKISH"), wxColour(234, 184, 184));
wxString name = wxTheColourDatabase->FindName(
                                         wxColour(234, 184, 184));
wxString color = wxTheColourDatabase->Find(name);
```

These are the available standard colors: aquamarine, black, blue, blue violet, brown, cadet blue, coral, cornflower blue, cyan, dark gray, dark green, dark olive green, dark orchid, dark slate blue, dark slate gray dark turquoise, dim gray, firebrick, forest green, gold, goldenrod, gray, green, green yellow, indian red, khaki, light blue, light gray, light steel blue, lime green, magenta, maroon, medium aquamarine, medium blue, medium forest green, medium goldenrod, medium orchid, medium sea green, medium slate blue, medium spring green, medium turquoise, medium violet red, midnight blue, navy, orange, orange red, orchid, pale green, pink, plum, purple, red, salmon, sea green, sienna, sky blue, slate blue, spring green, steel blue, tan, thistle, turquoise, violet, violet red, wheat, white, yellow, and yellow green.

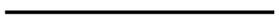
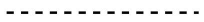
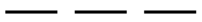
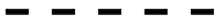
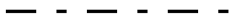
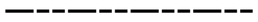
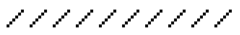
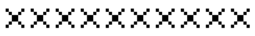
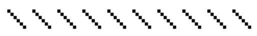
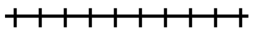
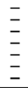
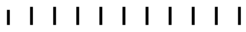
Smart_Ch05f.qxd 6/10/05 11:17 AM Page 142

wxPen

You define the current pen for the device context by passing a `wxPen` object to `SetPen`. The current pen defines the outline color, width, and style for subsequent drawing operations. `wxPen` has a low overhead, so you can create instances on the stack within your drawing code rather than storing them.

As well as a color and a width, a pen has a style, as described in Table 5-2. Hatch and stipple styles are not supported by the GTK+ port.

Table 5-2 *wxPen* Styles

Style	**Example**	**Description**
`wxSOLID`		Lines are drawn solid.
`wxTRANSPARENT`		Used when no pen drawing is desired.
`wxDOT`		The line is drawn dotted.
`wxLONG_DASH`		Draws with a long dashed style.
`wxSHORT_DASH`		Draws with a short dashed style. On Windows, this is the same as `wxLONG_SASH`.
`wxDOT_DASH`		Draws with a dot and a dash.
`wxSTIPPLE`		Uses a stipple bitmap, which is passed as the first constructor argument.
`wxUSER_DASH`		Uses user-specified dashes. See the reference manual for further information.
`wxBDIAGONAL_HATCH`		Draws with a backward-diagonal hatch.
`wxCROSSDIAG_HATCH`		Draws with a cross-diagonal hatch.
`wxFDIAGONAL_HATCH`		Draws with a forward-diagonal hatch.
`wxCROSS_HATCH`		Draws with a cross hatch.
`wxHORIZONTAL_HATCH`		Draws with a horizontal hatch.
`wxVERTICAL_HATCH`		Draws with a vertical hatch.

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 143

Call SetCap if you need to specify how the ends of thick lines should look: wxCAP_ROUND (the default) specifies rounded ends, wxCAP_PROJECTING specifies a square projection on either end, and wxCAP_BUTT specifies that the ends should be square and should not project.

You can call SetJoin to set the appearance where lines join. The default is wxJOIN_ROUND, where the corners are rounded. Other values are wxJOIN_BEVEL and wxJOIN_MITER.

There are some stock pens that you can use: wxRED_PEN, wxCYAN_PEN, wxGREEN_PEN, wxBLACK_PEN, wxWHITE_PEN, wxTRANSPARENT_PEN, wxBLACK_DASHED_PEN, wxGREY_PEN, wxMEDIUM_GREY_PEN, and wxLIGHT_GREY_PEN. These are pointers, so you'll need to dereference them when passing them to SetPen. There is also the object wxNullPen (an object, not a pointer), an uninitialized pen object that can be used to reset the pen in a device context.

Here are some examples of creating pens:

```
// A solid red pen
wxPen pen(wxColour(255, 0, 0), 1, wxSOLID);
wxPen pen(wxT("RED"), 1, wxSOLID);
wxPen pen = (*wxRED_PEN);
wxPen pen(*wxRED_PEN);
```

The last two examples use reference counting, so pen's internal data points to wxRED_PEN's data. Reference counting is used for all drawing objects, and it makes the assignment operator and copy constructor cheap operations, but it does mean that sometimes changes in one object affect the properties of another.

One way to reduce the amount of construction and destruction of pen objects without storing pen objects in your own classes is to use the global pointer wxThePenList to create and store the pens you need, for example:

```
wxPen* pen = wxThePenList->FindOrCreatePen(*wxRED, 1, wxSOLID);
```

The pen object will be stored in wxThePenList and cleaned up on application exit. Obviously, you should take care not to use this indiscriminately to avoid filling up memory with pen objects, and you also need to be aware of the reference counting issue mentioned previously. You can remove a pen from the list without deleting it by using RemovePen.

wxBrush

The current brush, specified with SetBrush, defines the fill color and style for drawing operations. You also specify the device context background color using a wxBrush, rather than with just a color. As with wxPen, wxBrush has a low overhead and can be created on the stack.

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 144

Pass a color and a style to the brush constructor. The style can be one of the values listed in Table 5-3.

Table 5-3 *wxBrush* Styles

Style	**Example**	**Description**
`wxSOLID`	■	Solid color is used.
`wxTRANSPARENT`		Used when no filling is desired.
`wxBDIAGONAL_HATCH`	▨	Draws with a backward-diagonal hatch.
`wxCROSSDIAG_HATCH`	▩	Draws with a cross-diagonal hatch.
`wxFDIAGONAL_HATCH`	▧	Draws with a forward-diagonal hatch.
`wxCROSS_HATCH`	▦	Draws with a cross hatch.
`wxHORIZONTAL_HATCH`	▤	Draws with a horizontal hatch.
`wxVERTICAL_HATCH`	▥	Draws with a vertical hatch.
`wxSTIPPLE`	★★★	Uses a stipple bitmap, which is passed as the first constructor argument.

You can use the following stock brushes: `wxBLUE_BRUSH`, `wxGREEN_BRUSH`, `wxWHITE BRUSH`, `wxBLACK_BRUSH`, `wxGREY_BRUSH`, `wxMEDIUM_GREY_BRUSH`, `wxLIGHT_GREY_BRUSH`, `wxTRANSPARENT_BRUSH`, `wxCYAN_BRUSH`, and `wxRED_BRUSH`. These are pointers. You can also use the `wxNullBrush` object (an uninitialized brush object).

Here are some examples of creating brushes:

```
// A solid red brush
wxBrush brush(wxColour(255, 0, 0), wxSOLID);
wxBrush brush(wxT("RED"), wxSOLID);
wxBrush brush = (*wxRED_BRUSH); // a cheap operation
wxBrush brush(*wxRED_BRUSH);
```

As with `wxPen`, `wxBrush` also has an associated list, `wxTheBrushList`, which you can use to cache brush objects:

```
wxBrush* brush = wxTheBrushList->FindOrCreateBrush(*wxBLUE, wxSOLID);
```

Use this with care to avoid proliferation of brush objects and side effects from reference counting. You can remove a brush from the list without deleting it by using `RemoveBrush`.

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 145

wxFont

You use font objects for specifying how text will appear when drawn on a device context. A font has the following properties:

The *point size* specifies the maximum height of the text in points (1/72 of an inch). wxWidgets will choose the closest match it can if the platform is not using scalable fonts.

The *font family* specifies one of a small number of family names, as described in Table 5-4. Specifying a family instead of an actual face name makes applications be more portable because you can't usually rely on a particular typeface being available on all platforms.

The *style* can be `wxNORMAL`, `wxSLANT`, or `wxITALIC`. `wxSLANT` may not be implemented for all platforms and fonts.

The *weight* is one of `wxNORMAL`, `wxLIGHT`, or `wxBOLD`.

A font's *underline* can be on (`true`) or off (`false`).

The *face name* is optional and specifies a particular typeface. If empty, a default typeface will be chosen from the family specification.

The optional *encoding* specifies the mapping between the character codes used in the program and the letters that are drawn onto the device context. Please see Chapter 16, "Writing International Applications," for more on this topic.

Table 5-4 Font Family Identifiers

Identifier	Example	Description
`wxFONTFAMILY_SWISS`	ABCDEFGabcdefg12345	A sans-serif font—often Helvetica or Arial depending on platform.
`wxFONTFAMILY_ROMAN`	ABCDEFGabcdefg12345	A formal, serif font.
`wxFONTFAMILY_SCRIPT`	ABCDEFGabcdefg12345	A handwriting font.
`wxFONTFAMILY_MODERN`	ABCDEFGabcdefg12345	A fixed pitch font, often Courier.
`wxFONTFAMILY_DECORATIVE`	ABCDEFGabcdefg12345	A decorative font.
`wxFONTFAMILY_DEFAULT`		wxWidgets chooses a default family.

You can create a font with the default constructor or by specifying the properties listed in Table 5-4.

There are some stock font objects that you can use: `wxNORMAL_FONT`, `wxSMALL_FONT`, `wxITALIC_FONT`, and `wxSWISS_FONT`. These have the size of the standard system font (`wxSYS_DEFAULT_GUI_FONT`), apart from `wxSMALL_FONT`, which is two points smaller. You can also use `wxSystemSettings::GetFont` to retrieve standard fonts.

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 146

To use a font object, pass it to `wxDC::SetFont` before performing text operations, in particular `DrawText` and `GetTextExtent`.

Here are some examples of font creation.

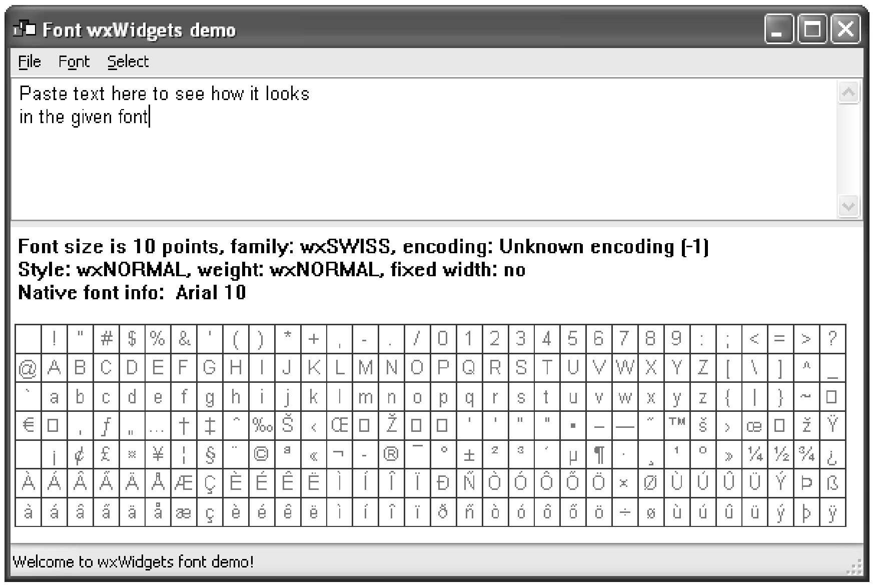
```
wxFont font(12, wxFONTFAMILY_ROMAN, wxITALIC, wxBOLD, false);
wxFont font(10, wxFONTFAMILY_SWISS, wxNORMAL, wxBOLD, true,
            wxT("Arial"), wxFONTENCODING_ISO8859_1));
wxFont font(wxSystemSettings::GetFont(wxSYS_DEFAULT_GUI_FONT));
```

`wxFont` has an associated list, `wxTheFontList`, which you can use to find a previously created font or add a new one:

```
wxFont* font = wxTheFontList->FindOrCreateFont(12, wxSWISS,
                                               wxNORMAL, wxNORMAL);
```

As with the pen and brush lists, use this with moderation because the fonts will be deleted only when the application exits. You can remove a font from the list without deleting it by using `RemoveFont`.

We'll see some examples of working with text and fonts later in the chapter. Also, you may like to play with the font demo in `samples/font` (see Figure 5-1). It lets you set a font to see how some text will appear, and you can change the font size and other properties.



```
Paste text here to see how it looks
in the given font
```

Font size is 10 points, family: wxSWISS, encoding: Unknown encoding (-1)
Style: wxNORMAL, weight: wxNORMAL, fixed width: no
Native font info: Arial 10

	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	□
€	□	‚	ƒ	„	…	†	‡	ˆ	‰	Š	‹	Œ	□	Ž	□	□	‘	’	“	”	•	–	—	˜	™	š	›	œ	□	ž	Ÿ
	¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬	-	®	¯	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Welcome to wxWidgets font demo!

Figure 5-1 wxWidgets font demo

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 147

wxPalette

A palette is a table, often with a size of 256, that maps index values to the red, green, and blue values of the display colors. It's normally used when the display has a very limited number of colors that need to be shared between applications. By setting a palette for a client device context, an application's color needs can be balanced with the needs of other applications. It is also used to map the colors of a low-depth bitmap to the available colors, and so `wxBitmap` has an optional associated `wxPalette` object.

Because most computers now have full-color displays, palettes are rarely needed. RGB colors specified in an application are mapped to the nearest display color with no need for a palette.

A `wxPalette` can be created by passing a size and three arrays (`unsigned char*`) for each of the red, green, and blue components. You can query the number of colors with `GetColoursCount`. To find the red, green, and blue values for a given index, use `GetRGB`, and you can find an index value for given red, green, and blue values with `GetPixel`.

Set the palette into a client, window, or memory device context with `wxDC::SetPalette`. For example, you can set the palette obtained from a low-depth `wxBitmap` you are about to draw, so the system knows how to map the index values to device colors. When using drawing functions that use `wxColour` with a device context that has a palette set, the RGB color will be mapped automatically to the palette index by the system, so choose a palette that closely matches the colors you will be using.

Another use for `wxPalette` is to query a `wxImage` or `wxBitmap` for the colors in a low-color image that was loaded from a file, such as a GIF. If there is an associated `wxPalette` object, it will give you a quick way to identify the unique colors in the original file, even though the image will have been converted to an RGB representation. Similarly, you can create and associate a palette with a `wxImage` that is to be saved in a reduced-color format. For example, the following fragment loads a PNG file and saves it as an 8-bit Windows bitmap file:

```
// Load the PNG
wxImage image(wxT("image.png"), wxBITMAP_TYPE_PNG);

// Make a palette
unsigned char* red = new unsigned char[256];
unsigned char* green = new unsigned char[256];
unsigned char* blue = new unsigned char[256];
for (size_t i = 0; i < 256; i ++)
{
    red[i] = green[i] = blue[i] = i;
}
wxPalette palette(256, red, green, blue);

// Set the palette and the BMP depth
image.SetPalette(palette);
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 148

```
image.SetOption(wxIMAGE_OPTION_BMP_FORMAT, wxBMP_8BPP_PALETTE);

// Save the file
image.SaveFile(wxT("image.bmp"), wxBITMAP_TYPE_BMP);
```

More realistic code would "quantize" the image to reduce the number of colors; see "Color Reduction" in Chapter 10, "Programming with Images," for use of the `wxQuantize` class to do this.

wxWidgets defines a null palette object, `wxNullPalette`.

DEVICE CONTEXT DRAWING FUNCTIONS

In this section, we'll take a closer look at how we draw on device contexts. The major functions are summarized in Table 5-5. We cover most of them in the following sections, and you can also find more details in the reference manual.

Table 5-5 Device Context Functions

`Blit`	Copies from one device context to another. You can specify how much of the original to draw, where drawing should start, the logical function to use, and whether to use a mask if the source is a memory device context.
`Clear`	Fills the device context with the current background brush.

`SetClippingRegion`
`DestroyClippingRegion`
`GetClippingBox`
	Sets and destroys the clipping region, which restricts drawing to a specified area. The clipping region can be specified as a rectangle or a `wxRegion`. Use `GetClippingBox` to get the rectangle surrounding the current clipping region.

`DrawArc` `DrawEllipticArc`	Draws an arc or elliptic arc using the current pen and brush.
`DrawBitmap` `DrawIcon`	Draws a `wxBitmap` or `wxIcon` at the specified location. The bitmap may have a mask to specify transparency.
`DrawCircle`	Draws a circle using the current pen and brush.
`DrawEllipse`	Draws an ellipse using the current pen and brush.
`DrawLine` `DrawLines`	Draws a line or number of lines using the current pen. The last point of the line is not drawn.
`DrawPoint`	Draws a point using the current pen.
`DrawPolygon` `DrawPolyPolygon`	`DrawPolygon` draws a filled polygon using an array of points or list of pointers to points, adding an optional offset coordinate. wxWidgets automatically closes the first and last points. `DrawPolyPolygon` draws one or more polygons at once, which can be a more efficient operation on some platforms.

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 149

`DrawRectangle` `DrawRoundedRectangle`	Draws a rectangle or rounded rectangle using the current pen and brush.
`DrawText` `DrawRotatedText`	Draws a text string, or rotated text string, at the specified point using the current text font and the current text foreground and background colors.
`DrawSpline`	Draws a spline between all given control points, using the current pen.
`FloodFill`	Flood fills the device context starting from the given point, using the current brush color.
`Ok`	Returns `true` if the device context is OK to use.
`SetBackground` `GetBackground`	Sets and gets the background brush used in `Clear` and in functions that use a complex logical function. The default is `wxTRANSPARENT_BRUSH`.
`SetBackgroundMode` `GetBackgroundMode`	Sets and gets the background mode for drawing text: `wxSOLID` or `wxTRANSPARENT`. Normally, you will want to set the mode to `wxTRANSPARENT` (the default) so the existing background will be kept when drawing text.
`SetBrush` `GetBrush`	Sets and gets the brush to be used to fill shapes in subsequent drawing operations. The initial value of the brush is undefined.
`SetPen` `GetPen`	Sets and gets the pen to be used to draw the outline of shapes in subsequent drawing operations. The initial value of the pen is undefined.
`SetFont` `GetFont`	Sets and gets the font to be used in `DrawText`, `DrawRotatedText`, and `GetTextExtent` calls. The initial value of the font is undefined.
`SetPalette` `GetPalette`	Sets and gets `wxPalette` object mapping index values to RGB colors.

`SetTextForeground`
`GetTextForeground`
`SetTextBackground`
`GetTextBackground`
	Sets and gets the color to be used for text foreground and background. The defaults are black and white, respectively.

`SetLogicalFunction` `GetLogicalFunction`	The logical function determines how a source pixel from a pen or brush color, or source device context if using `Blit`, combines with a destination pixel in the current device context. The default is `wxCOPY`, which simply draws with the current color.
`GetPixel`	Returns the color at the given point. This is not implemented for `wxPostScriptDC` and `wxMetafileDC`.
`GetTextExtent` `GetPartialTextExtents`	Returns metrics for a given text string.
`GetSize` `GetSizeMM`	Returns the dimensions of the device in device units or millimeters.

(*continues*)

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 150

Table 5-5 Device Context Functions (*Continued*)

`StartDoc` `EndDoc`	Starts and ends a document. This is only applicable to printer device contexts. When `StartDoc` is called, a message will be shown as the document is printed, and `EndDoc` will hide the message box.
`StartPage` `EndPage`	Starts and ends a page. This is only applicable to printer device contexts.

`DeviceToLogicalX`
`DeviceToLogicalXRel`
`DeviceToLogicalY`
`DeviceToLogicalYRel`
	Converts device coordinates to logical coordinates, either absolute (for positions) or relative (for widths and heights).

`LogicalToDeviceX`
`LogicalToDeviceXRel`
`LogicalToDeviceY`
`LogicalToDeviceYRel`
	Converts logical coordinates to device coordinates, either absolute (for positions) or relative (for widths and heights).

`SetMapMode` `GetMapMode`	As described earlier, this determines (along with `SetUserScale`) how logical units are converted to device units.
`SetAxisOrientation`	Sets the *x*- and *y*-axis orientation: the direction from lowest to highest values on the axis. The default orientation is to have the *x*-axis from left to right (`true`) and the *y*-axis from top to bottom (`false`).
`SetDeviceOrigin` `GetDeviceOrigin`	Sets and gets the device origin. You can use this to place a graphic in a particular place on a page, for example.
`SetUserScale` `GetUserScale`	Sets and gets the scale to be applied when converting from logical units to device units.

Drawing Text

The way text is drawn on a device context with `DrawText` is determined by the current font, the background mode (transparent or solid drawing), and the text foreground and background colors. If the background mode is `wxSOLID`, the area behind the text will be drawn in the current background color, and if `wxTRANSPARENT`, the text will be drawn without disturbing the background.

Pass a string and either two integers or a `wxPoint` to `DrawText`. The text will be drawn with the given location at the very top-left of the string. Here's a simple example of drawing a text string:

```
// Draw a text string on a window at the given point
void DrawTextString(wxDC& dc, const wxString& text,
                    const wxPoint& pt)
{
    wxFont font(12, wxFONTFAMILY_SWISS, wxNORMAL, wxBOLD);
    dc.SetFont(font);
    dc.SetBackgroundMode(wxTRANSPARENT);
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 151

```
    dc.SetTextForeground(*wxBLACK);
    dc.SetTextBackground(*wxWHITE);
    dc.DrawText(text, pt);
}
```

You can also use the device context function `DrawRotatedText` to draw text at an angle by specifying the angle in degrees as the last parameter. The following code draws text at 45-degree increments, and the result is illustrated in Figure 5-2.

```
wxFont font(20, wxFONTFAMILY_SWISS, wxNORMAL, wxNORMAL);

dc.SetFont(font);
dc.SetTextForeground(wxBLACK);
dc.SetBackgroundMode(wxTRANSPARENT);

for (int angle = 0; angle < 360; angle += 45)
    dc.DrawRotatedText(wxT("Rotated text..."), 300, 300, angle);
```

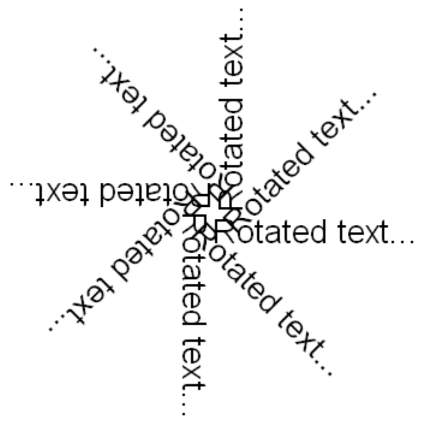


Figure 5-2 Drawing rotated text

On Windows, only TrueType fonts can be drawn rotated. Be aware that the stock object `wxNORMAL_FONT` is not TrueType.

Often, you'll need to find out how much space text will take on a device context, which you can do by passing `wxCoord` (integer) pointers to the function `GetTextExtent`. Its prototype is

```
void GetTextExtent(const wxString& string,
    wxCoord* width, wxCoord* height,
    wxCoord* descent = NULL, wxCoord* externalLeading = NULL,
    wxFont* font = NULL);
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 152

The default arguments mean that you can call it just to find the overall width and height the string occupies, or you can pass extra arguments to get further text dimensions. If you imagine the bottoms of the characters sitting on a baseline, the *descent* is how far below the baseline the characters extend. The letter “g,” for example, extends below the baseline. *External leading* is the space between the descent of one line and the top of the line below. Finally, you can provide a font to be used in place of the current device context font.

Here’s code that uses `GetTextExtent` to center a string on a window:

```
void CenterText(const wxString& text, wxDC& dc, wxWindow* win)
{
    // Set font, background mode for drawing text,
    // and text color
    dc.SetFont(*wxNORMAL_FONT);
    dc.SetBackgroundMode(wxTRANSPARENT);
    dc.SetTextForeground(*wxRED);

    // Get window and text dimensions
    wxSize sz = win->GetClientSize();
    wxCoord w, h;
    dc.GetTextExtent(text, & w, & h);

    // Center the text on the window, but never
    // draw at a negative position.
    int x = wxMax(0, (sz.x - w)/2);
    int y = wxMax(0, (sz.y - h)/2);

    dc.DrawText(msg, x, y);
}
```

You can also use `GetPartialTextExtents` to retrieve the width of each character, passing a `wxString` and a `wxArrayInt` reference to receive the character width values. If you need accurate information about individual character widths, this can be quicker on some platforms than calling `GetTextExtent` for each character.

Drawing Lines and Shapes

The simpler drawing primitives include points, lines, rectangles, circles, and ellipses. The current pen determines the line or outline color, and the brush determines the fill color. For example:

```
void DrawSimpleShapes(wxDC& dc)
{
    // Set line color to black, fill color to green
    dc.SetPen(wxPen(*wxBLACK, 2, wxSOLID));
    dc.SetBrush(wxBrush(*wxGREEN, wxSOLID));

    // Draw a point
    dc.DrawPoint(5, 5);
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 153

```
    // Draw a line
    dc.DrawLine(10, 10, 100, 100);

    // Draw a rectangle at (50, 50) with size (150, 100)
    // and hatched brush
    dc.SetBrush(wxBrush(*wxBLACK, wxCROSS_HATCH));
    dc.DrawRectangle(50, 50, 150, 100);

    // Set a red brush
    dc.SetBrush(*wxRED_BRUSH);

    // Draw a rounded rectangle at (150, 20) with size (100, 50)
    // and corner radius 10
    dc.DrawRoundedRectangle(150, 20, 100, 50, 10);

    // Draw another rounded rectangle with no border
    dc.SetPen(*wxTRANSPARENT_PEN);
    dc.SetBrush(wxBrush(*wxBLUE));
    dc.DrawRoundedRectangle(250, 80, 100, 50, 10);

    // Set a black pen and black brush
    dc.SetPen(wxPen(*wxBLACK, 2, wxSOLID));
    dc.SetBrush(*wxBLACK);

    // Draw a circle at (100, 150) with radius 60
    dc.DrawCircle(100, 150, 60);

    // Set a white brush
    dc.SetBrush(*wxWHITE);

    // Draw an ellipse that fills the given rectangle
    dc.DrawEllipse(wxRect(120, 120, 150, 50));
}
```

This produces the graphic in Figure 5-3.

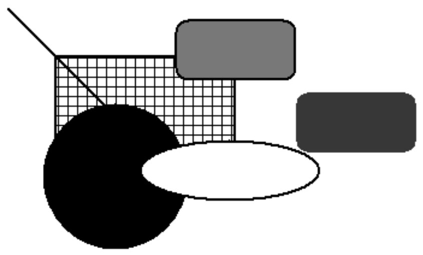


Figure 5-3 Drawing simple shapes

Note that by convention, the last point of a line is not drawn.

To draw a circular arc, use `DrawArc`, taking a starting point, end point, and center point. The arc is drawn counterclockwise from the starting point to the end. For example:

```
// Draw a cup-shaped arc
int x = 10, y = 200, radius = 20;
dc.DrawArc(x-radius, y, x + radius, y, x, y);
```

This produces the arc shown in Figure 5-4.

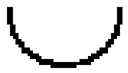


Figure 5-4 A circular arc

For an elliptic arc, `DrawEllipticArc` takes the position and size of a rectangle that contains the arc, plus the start and end of the arc in degrees specified from the three o'clock position from the center of the rectangle. If the start and end points are the same, a complete ellipse will be drawn. The following code draws the arc shown in Figure 5-5.

```
// Draws an elliptical arc within a rectangle at (10, 100),
// size 200x40. Arc extends from 270 to 420 degrees.
dc.DrawEllipticArc(10, 100, 200, 40, 270, 420);
```

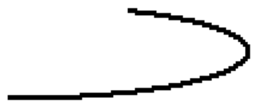


Figure 5-5 An elliptical arc

If you need to draw a lot of lines quickly, `DrawLines` can be more efficient than using `DrawLine` multiple times. The following example draws lines between ten points, at an offset of (100, 100).

```
wxPoint points[10];
for (size_t i = 0; i < 10; i++)
{
  pt.x = i*10; pt.y = i*20;
}
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 155

```
int offsetX = 100;
int offsetY = 100;

dc.DrawLines(10, points, offsetX, offsetY);
```

`DrawLines` does not fill the area surrounded by the lines. You can draw a filled shape with an arbitrary number of sides using `DrawPolygon`, and several of them with `DrawPolyPolygon`. `DrawPolygon` takes a point count, an array of points, optional offsets to add to the points, and an optional fill style: `wxODDEVEN_RULE`, the default, or `wxWINDING_RULE`. `DrawPolygonPolygon` additionally takes an array of integers that specifies the number of points to be used for each polygon.

The following code demonstrates how to draw polygons and poly-polygons, with the result shown in Figure 5-6.

```
void DrawPolygons(wxDC& dc)
{
    wxBrush brushHatch(*wxRED, wxFDIAGONAL_HATCH);
    dc.SetBrush(brushHatch);

    wxPoint star[5];
    star[0] = wxPoint(100, 60);
    star[1] = wxPoint(60, 150);
    star[2] = wxPoint(160, 100);
    star[3] = wxPoint(40, 100);
    star[4] = wxPoint(140, 150);

    dc.DrawPolygon(WXSIZEOF(star), star, 0, 30);
    dc.DrawPolygon(WXSIZEOF(star), star, 160, 30, wxWINDING_RULE);

    wxPoint star2[10];
    star2[0] = wxPoint(0, 100);
    star2[1] = wxPoint(-59, -81);
    star2[2] = wxPoint(95, 31);
    star2[3] = wxPoint(-95, 31);
    star2[4] = wxPoint(59, -81);
    star2[5] = wxPoint(0, 80);
    star2[6] = wxPoint(-47, -64);
    star2[7] = wxPoint(76, 24);
    star2[8] = wxPoint(-76, 24);
    star2[9] = wxPoint(47, -64);
    int count[2] = {5, 5};

    dc.DrawPolyPolygon(WXSIZEOF(count), count, star2, 450, 150);
}
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 156

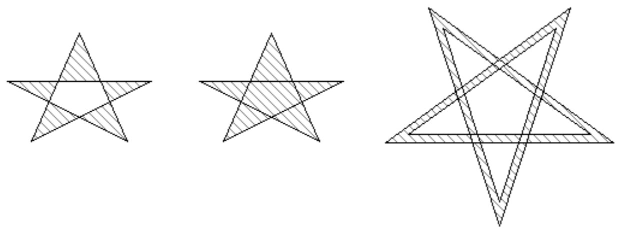


Figure 5-6 Drawing polygons

Drawing Splines

`DrawSpline` lets you draw a curve known as a "spline" between multiple points. There is a version for three points, and a version for an arbitrary number of points, both illustrated in this example code:

```
// Draw 3-point sline
dc.DrawSpline(10, 100, 200, 200, 50, 230);

// Draw 5-point spline
wxPoint star[5];
star[0] = wxPoint(100, 60);
star[1] = wxPoint(60, 150);
star[2] = wxPoint(160, 100);
star[3] = wxPoint(40, 100);
star[4] = wxPoint(140, 150);
dc.DrawSpline(WXSIZEOF(star), star);
```

This produces the two splines illustrated in Figure 5-7.

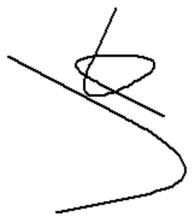


Figure 5-7 Drawing splines

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 157

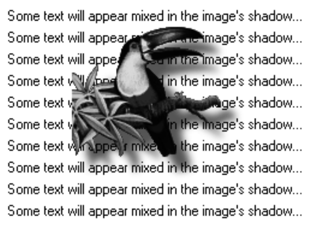
Drawing Bitmaps

There are two main ways of drawing bitmaps on a device context: `DrawBitmap` and `Blit`. `DrawBitmap` is a simplified form of `Blit`, and it takes a bitmap, a position, and a boolean flag specifying transparent drawing. The transparency can be either a simple mask or an alpha channel (which offers translucency), depending on how the bitmap was loaded or created. The following code loads an image with an alpha channel and draws it over lines of text.

```
wxString msg = wxT("Some text will appear mixed in the image's
shadow...");
int y = 75;
for (size_t i = 0; i < 10; i++)
{
    y += dc.GetCharHeight() + 5;
    dc.DrawText(msg, 200, y);
}

wxBitmap bmp(wxT("toucan.png"), wxBITMAP_TYPE_PNG);
dc.DrawBitmap(bmp, 250, 100, true);
```

This produces the drawing in Figure 5-8, where the shadows in the bitmap appear to partially obscure the text underneath.



```
Some text will appear mixed in the image's shadow...
Some text will appear          mage's shadow...
Some text will appe         image's shadow...
Some text w          in the image's shadow...
Some text w               ge's shadow...
Some text              n the image's shadow...
Some text w    ppe  r mixe   the image's shadow...
Some text will appear mixed in the image's shadow...
Some text will appear mixed in the image's shadow...
Some text will appear mixed in the image's shadow...
```

Figure 5-8 Drawing with transparency

The `Blit` function is more flexible and enables you to copy a specific portion of a source device context onto a destination device context. This is its prototype:

```
bool Blit(wxCoord destX, wxCoord destY,
          wxCoord width, wxCoord height, wxDC* dcSource,
          wxCoord srcX, wxCoord srcY,
          int logicalFunc = wxCOPY,
          bool useMask = false,
          wxCoord srcMaskX = -1, wxCoord srcMaskY = -1);
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 158

This code copies an area from a source device context `dcSource` to the destination (the object that the function is operating on). An area of specified `width` and `height` is drawn starting at the position (`destX`, `destY`) on the destination surface, taken from the position (`srcX`, `srcY`) on the source. The logical function `logicalFunc` is usually `wxCOPY`, which means the bits are transferred from source to destination with no transformation. Not all platforms support a logical function other than `wxCOPY`. For more information, please see "Logical Functions" later in this chapter.

The last three parameters are used only when the source device context is a `wxMemoryDC` with a transparent bitmap selected into it. `useMask` specifies whether transparent drawing is used, and `srcMaskX` and `srcMaskY` enable the bitmap's mask to start from a different position than the main bitmap start position.

The following example loads a small pattern into a bitmap and uses `Blit` to fill a larger destination bitmap, with transparency if available.

```
wxMemoryDC dcDest;
wxMemoryDC dcSource;

int destWidth = 200, destHeight = 200;

// Create the destination bitmap
wxBitmap bitmapDest(destWidth, destHeight);

// Load the pattern bitmap
wxBitmap bitmapSource(wxT("pattern.png"), wxBITMAP_TYPE_PNG);

int sourceWidth = bitmapSource.GetWidth();
int sourceHeight = bitmapSource.GetHeight();

// Clear the destination background to white
dcDest.SelectObject(bitmapDest);
dcDest.SetBackground(*wxWHITE_BRUSH);
dcDest.Clear();

dcSource.SelectObject(bitmapSource);

// Tile the smaller bitmap onto the larger bitmap
for (int i = 0; i < destWidth; i += sourceWidth)
    for (int j = 0; j < destHeight; j += sourceHeight)
    {
        dcDest.Blit(i, j, sourceWidth, sourceHeight,
                    & dcSource, 0, 0, wxCOPY, true);
    }

// Tidy up
dcDest.SelectBitmap(wxNullBitmap);
dcSource.SelectBitmap(wxNullBitmap);
```

You can also draw icons directly, with `DrawIcon`. This operation always takes transparency into account. For example:

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 159

```
#include "file.xpm"

wxIcon icon(file_xpm);
dc.DrawIcon(icon, 20, 30);
```

Filling Arbitrary Areas

`FloodFill` can be used to fill an arbitrary area of a device context up to a color boundary. Pass a starting point, a color for finding the flood area boundary, and a style to indicate how the color parameter should be used. The device context will be filled with the current brush color.

The following example draws a green rectangle with a red border and fills it with black, followed by blue.

```
// Draw a green rectangle outlines in red
dc.SetPen(*wxRED_PEN);
dc.SetBrush(*wxGREEN_BRUSH);

dc.DrawRectangle(10, 10, 100, 100);
dc.SetBrush(*wxBLACK_BRUSH);

// Now fill the green area with black (while green is found)
dc.FloodFill(50, 50, *wxGREEN, wxFLOOD_SURFACE);
dc.SetBrush(*wxBLUE_BRUSH);

// Then fill with blue (until red is encountered)
dc.FloodFill(50, 50, *wxRED, wxFLOOD_BORDER);
```

The function may fail if it cannot find the color specified, or the point is outside the clipping region. `FloodFill` won't work with printer device contexts, or with `wxMetafileDC`.

Logical Functions

The current *logical function* determines how a source pixel (from a pen or brush color, or source device context if using `Blit`) combines with a destination pixel in the current device context. The default is `wxCOPY`, which simply draws with the current color. The others combine the current color and the background using a logical operation. `wxINVERT` is commonly used for drawing rubber bands or moving outlines because with this operation drawing a shape the second time erases the shape.

The following example draws a dotted line using `wxINVERT` and then erases it before restoring the normal logical function.

```
wxPen pen(*wxBLACK, 1, wxDOT);
dc.SetPen(pen);

// Invert pixels
dc.SetLogicalFunction(wxINVERT);
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 160

```
dc.DrawLine(10, 10, 100, 100);

// Invert again, rubbing it out
dc.DrawLine(10, 10, 100, 100);

// Restore to normal drawing
dc.SetLogicalFunction(wxCOPY);
```

Another use for logical functions is to combine images to create new images. For example, here's one method for creating transparent jigsaw puzzle pieces out of an image. First, draw a black outline of each shape on a white bitmap, using a grid of standard (but randomized) puzzle edges. Then, for each piece, flood-fill the outline to create a black puzzle shape on a white background. `Blit` the corresponding area of the puzzle image onto this template bitmap with the `wxAND_REVERSE` function to mask out the unwanted parts of the puzzle, leaving the "stamped out" puzzle piece on a black background. This can be made into a transparent `wxBitmap` by converting to a `wxImage`, setting black as the image mask color, and converting back to a transparent `wxBitmap`, which can be drawn appropriately. (Note that this technique depends on there being no black in the puzzle image, or else holes will appear in the puzzle pieces.)

Table 5-6 shows the logical function values and their meanings.

Table 5-6 Logical Functions

Logical Function	**Meaning (*src* = source, *dst* = destination)**
`wxAND`	src AND dst
`wxAND_INVERT`	(NOT src) AND dst
`wxAND_REVERSE`	src AND (NOT dst)
`wxCLEAR`	0
`wxCOPY`	src
`wxEQUIV`	(NOT src) XOR dst
`wxINVERT`	NOT dst
`wxNAND`	(NOT src) OR (NOT dst)
`wxNOR`	(NOT src) AND (NOT dst)
`wxNO_OP`	dst
`wxOR`	src OR dst
`wxOR_INVERT`	(NOT src) OR dst
`wxOR_REVERSE`	src OR (NOT dst)
`wxSET`	1
`wxSRC_INVERT`	NOT src
`wxXOR`	src XOR dst

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 161

USING THE PRINTING FRAMEWORK

As we've seen, wxPrinterDC can be created and used directly. However, a more flexible method is to use the wxWidgets printing framework to "drive" printing. The main task for the developer is to derive a new class from wxPrintout, overriding functions that specify how to print a page (OnPrintPage), how many pages there are (GetPageInfo), document setup (OnPreparePrinting), and so on. The wxWidgets printing framework will show the print dialog, create the printer device context, and call appropriate wxPrintout functions when appropriate. The same printout class can be used for both printing and preview.

To start printing, a wxPrintout object is passed to a wxPrinter object, and Print is called to kick off the printing process, showing a print dialog before printing the pages specified by the layout object and the user. For example:

```
// A global object storing print settings
wxPrintDialogData g_printDialogData;

// Handler for Print menu item
void MyFrame::OnPrint(wxCommandEvent& event)
{

    wxPrinter printer(& g_printDialogData);
    MyPrintout printout(wxT("My printout"));

    if (!printer.Print(this, &printout, true))
    {
        if (wxPrinter::GetLastError() == wxPRINTER_ERROR)
            wxMessageBox(wxT("There was a problem printing.\nPerhaps your
current printer is not set correctly?"), wxT("Printing"), wxOK);
        else
            wxMessageBox(wxT("You cancelled printing"),
                         wxT("Printing"), wxOK);
    }
    else
    {
        (*g_printDialogData) = printer.GetPrintDialogData();
    }
}
```

Because the Print function returns only after all pages have been rendered and sent to the printer, the printout object can be created on the stack.

The wxPrintDialogData class stores data related to the print dialog, such as the pages the user selected for printing and the number of copies to be printed. It's a good idea to keep a global wxPrintDialogData object in your application to store the last settings selected by the user. You can pass a pointer to this data to wxPrinter to be used in the print dialog, and then if printing is successful, copy the settings back from wxPrinter to your global object, as in the previous example. (In a real application, g_printDialogData would probably be

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 162

a data member of your application class.) See Chapter 8, “Using Standard Dialogs,” for more about print and page dialogs and how to use them.

To preview the document, create a `wxPrintPreview` object, passing two printout objects to it: one for the preview and one to use for printing if the user requests it. You can also pass a `wxPrintDialogData` object so that the preview picks up settings that the user chose earlier. Then pass the preview object to `wxPreviewFrame`, call the frame’s `Initialize` function, and show the frame. For example:

```
// Handler for Preview menu item
void MyFrame::OnPreview(wxCommandEvent& event)
{
    wxPrintPreview *preview = new wxPrintPreview(
                             new MyPrintout, new MyPrintout,
                             & g_printDialogData);
    if (!preview->Ok())
    {
        delete preview;
        wxMessageBox(wxT("There was a problem previewing.\nPerhaps your
current printer is not set correctly?"),
                     wxT("Previewing"), wxOK);
        return;
    }

    wxPreviewFrame *frame = new wxPreviewFrame(preview, this,
                             wxT("Demo Print Preview"));
    frame->Centre(wxBOTH);
    frame->Initialize();
    frame->Show(true);
}
```

When the preview frame is initialized, it disables all other top-level windows in order to avoid actions that might cause the document to be edited after the print or preview process has started. Closing the frame automatically destroys the two printout objects. Figure 5-9 shows the print preview window, with a control bar along the top providing page navigation, printing, and zoom control.

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 163

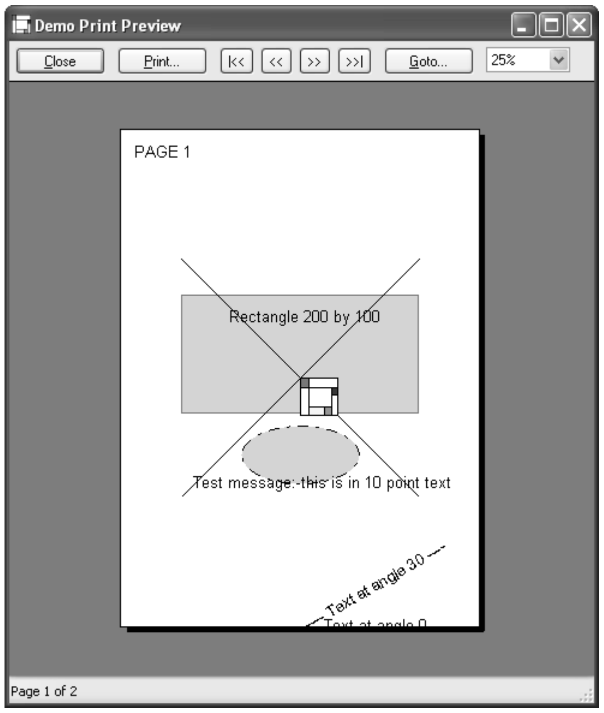


Figure 5-9 Print preview window

More on *wxPrintout*

When creating a printout object, the application can pass an optional title that will appear in the print manager under some operating systems. You will need to provide at least `GetPageInfo`, `HasPage`, and `OnPrintPage`, but you can override any of the other methods below as well.

`GetPageInfo` should be overridden to return `minPage`, `maxPage`, `pageFrom`, and `pageTo`. The first two integers represent the range supported by this printout object for the current document, and the second two integers represent a user selection (not currently used by wxWidgets). The default values for `minPage` and `maxPage` are `1` and `32,000`, respectively. However, the printout will stop printing if `HasPage` returns `false`. Typically, your `OnPreparePrinting` function will calculate the values returned by `GetPageInfo` and will look something like this:

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 164

```
void MyPrintout::GetPageInfo(int *minPage, int *maxPage,
                             int *pageFrom, int *pageTo)
{
    *minPage = 1; *maxPage = m_numPages;
    *pageFrom = 1; *pageTo = m_numPages;
}
```

HasPage must return false if the argument is outside the current page range. Often its implementation will look like this, where m_numPages has been calculated in OnPreparePrinting:

```
bool MyPrintout::HasPage(int pageNum)
{
    return (pageNum >= 1 && pageNum <= m_numPages);
}
```

OnPreparePrinting is called before the print or preview process commences, and overriding it enables the application to do various setup tasks, including calculating the number of pages in the document. OnPreparePrinting can call wxPrintout functions such as GetDC, GetPageSizeMM, IsPreview, and so on to get the information it needs.

OnBeginDocument is called with the start and end page numbers when each document copy is about to be printed, and if overridden, it must call the base wxPrintout::OnBeginDocument function. Similarly, wxPrintout::OnEndDocument must be called if overridden.

OnBeginPrinting is called once for the printing cycle, regardless of the number of copies, and OnEndPrinting is called at the end.

OnPrintPage is passed a page number, and the application should override it to return true if the page was successfully printed (returning false cancels the print job). This function will use wxPrintout::GetDC to get the device context to draw on.

The following are the utility functions you can use in your overridden functions, and they do not need to be overridden.

IsPreview can be called to determine whether this is a real print task or a preview.

GetDC returns a suitable device context for the current task. When printing, a wxPrinterDC will be returned, and when previewing, a wxMemoryDC will be returned because a preview is rendered into a bitmap via a memory device context.

GetPageSizeMM returns the size of the printer page in millimeters, whereas GetPageSizePixels returns the size in pixels (the maximum resolution of the printer). For a preview, this will not be the same as the size returned by wxDC::GetSize, which will return the preview bitmap size.

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 165

`GetPPIPrinter` returns the number of pixels per logical inch for the current device context, and `GetPPIScreen` returns the number of pixels per logical inch of the screen.

Scaling for Printing and Previewing

When drawing on a window, you probably don't concern yourself about scaling your graphics because displays tend to have similar resolutions. However, there are several factors to take into account when drawing to a printer:

- ☞ You need to scale and position graphics to fit the width of the page, and break the graphics into pages if necessary.
- ☞ Fonts are based on screen resolution, so when drawing text, you need to set a scale so that the printer device context matches the screen resolution. Dividing the printer resolution (`GetPPIPrinter`) by the screen resolution (`GetPPIScreen`) can give a suitable scaling factor for drawing text.
- ☞ When rendering the preview, wxWidgets uses a `wxMemoryDC` to draw into a bitmap. The size of the bitmap (returned by `wxDC::GetSize`) depends on the zoom scale, and an extra scale factor must be calculated to deal with this. Divide the size returned by `GetSize` by the actual page size returned by `GetPageSizePixels` to get this scale factor. This value should be multiplied by any other scale you calculated.

You can use `wxDC::SetUserScale` to let the device context perform the scaling for subsequent graphics operations and `wxDC::SetDeviceOrigin` to set the origin (for example, to center a graphic on a page). You can keep calling these scaling and device origin functions for different parts of your graphics, on the same page if necessary.

The wxWidgets sample in `samples/printing` shows how to do scaling. The following example shows a function adapted from the printing sample, which scales and positions a 200×200 pixel graphic on a printer or preview device context.

```
void MyPrintout::DrawPageOne(wxDC *dc)
{
    // You might use THIS code if you were scaling
    // graphics of known size to fit on the page.

    // We know the graphic is 200x200. If we didn't know this,
    // we'd need to calculate it.
    float maxX = 200;
    float maxY = 200;

    // Let's have at least 50 device units margin
    float marginX = 50;
    float marginY = 50;
    // Add the margin to the graphic size
    maxX += (2*marginX);
    maxY += (2*marginY);

    // Get the size of the DC in pixels
    int w, h;
    dc->GetSize(&w, &h);

    // Calculate a suitable scaling factor
    float scaleX=(float)(w/maxX);
    float scaleY=(float)(h/maxY);

    // Use x or y scaling factor, whichever fits on the DC
    float actualScale = wxMin(scaleX,scaleY);

    // Calculate the position on the DC for centring the graphic
    float posX = (float)((w - (200*actualScale))/2.0);
    float posY = (float)((h - (200*actualScale))/2.0);

    // Set the scale and origin
    dc->SetUserScale(actualScale, actualScale);
    dc->SetDeviceOrigin( (long)posX, (long)posY );

    // Now do the actual drawing
    dc.SetBackground(*wxWHITE_BRUSH);
    dc.Clear();
    dc.SetFont(wxGetApp().m_testFont);

    dc.SetBackgroundMode(wxTRANSPARENT);

    dc.SetBrush(*wxCYAN_BRUSH);
    dc.SetPen(*wxRED_PEN);

    dc.DrawRectangle(0, 30, 200, 100);

    dc.DrawText( wxT("Rectangle 200 by 100"), 40, 40);

    dc.SetPen( wxPen(*wxBLACK,0,wxDOT_DASH) );
    dc.DrawEllipse(50, 140, 100, 50);
    dc.SetPen(*wxRED_PEN);

    dc.DrawText( wxT("Test message: this is in 10 point text"),
                 10, 180);
}
```

In this code, we simply use `wxDC::GetSize` to get the preview or printer resolution so we can fit the graphic on the page. In this example, we're not interested in the points-per-inch printer resolution, as we might be if we were drawing text or lines of a specific length in millimeters, because the graphic doesn't have to be a precise size: it's just scaled to fit the available space.

Next, we'll show code that prints text at a size to match how it appears on the screen and that also draws lines that have a precise length, rather than simply being scaled to fit.

```
void MyPrintout::DrawPageTwo(wxDC *dc)
{
    // You might use THIS code to set the printer DC to roughly
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 167

```
    // reflect the screen text size. This page also draws lines of
    // actual length 5cm on the page.

    // Get the logical pixels per inch of screen and printer
    int ppiScreenX, ppiScreenY;
    GetPPIScreen(&ppiScreenX, &ppiScreenY);
    int ppiPrinterX, ppiPrinterY;
    GetPPIPrinter(&ppiPrinterX, &ppiPrinterY);

    // This scales the DC so that the printout roughly represents the
    // the screen scaling.
    float scale = (float)((float)ppiPrinterX/(float)ppiScreenX);

    // Now we have to check in case our real page size is reduced
    // (e.g. because we're drawing to a print preview memory DC)
    int pageWidth, pageHeight;
    int w, h;
    dc->GetSize(&w, &h);
    GetPageSizePixels(&pageWidth, &pageHeight);

    // If printer pageWidth == current DC width, then this doesn't
    // change. But w might be the preview bitmap width,
    // so scale down.
    float overallScale = scale * (float)(w/(float)pageWidth);
    dc->SetUserScale(overallScale, overallScale);

    // Calculate conversion factor for converting millimetres into
    // logical units.
    // There are approx. 25.4 mm to the inch. There are ppi
    // device units to the inch. Therefore 1 mm corresponds to
    // ppi/25.4 device units. We also divide by the
    // screen-to-printer scaling factor, because we need to
    // unscale to pass logical units to DrawLine.

    // Draw 50 mm by 50 mm L shape
    float logUnitsFactor = (float)(ppiPrinterX/(scale*25.4));
    float logUnits = (float)(50*logUnitsFactor);
    dc->SetPen(* wxBLACK_PEN);
    dc->DrawLine(50, 250, (long)(50.0 + logUnits), 250);
    dc->DrawLine(50, 250, 50, (long)(250.0 + logUnits));

    dc->SetBackgroundMode(wxTRANSPARENT);
    dc->SetBrush(*wxTRANSPARENT_BRUSH);

    dc->SetFont(wxGetApp().m_testFont);

    dc->DrawText(wxT("Some test text"), 200, 300 );
}
```

Printing Under Unix with GTK+

Unlike Mac OS X and Windows, Unix does not provide a standard way to display text and graphics onscreen and print it using the same API. Instead, screen display is done via the X11 library (via GTK+ and wxWidgets), whereas printing has to be done by sending a file of PostScript commands to the printer. Fonts are particularly tricky to handle; until recently, only a small number of applications have offered WYSIWYG (What You See Is What You

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 168

Get) under Unix. In the past, wxWidgets offered its own printing implementation using PostScript that never fully matched the screen display.

From version 2.8, the GNOME Free Software Desktop Project provides printing support through the `libgnomeprint` and `libgnomeprintui` libraries by which most printing problems are solved. Beginning with version 2.5.4, the GTK+ port of wxWidgets can make use of these libraries if wxWidgets is configured accordingly and if the libraries are present. You need to configure wxWidgets with the `--with-gnomeprint` switch, which will cause your application to search for the GNOME print libraries at runtime. If they are found, printing will be done through these; otherwise, the application will fall back to the old PostScript printing code. Note that the application will not require the GNOME print libraries to be installed in order to run (there is no dependency on these libraries).

3D Graphics with *wxGLCanvas*

It's worth mentioning that wxWidgets comes with the capability of drawing 3D graphics, thanks to OpenGL and `wxGLCanvas`. You can use it with the OpenGL clone Mesa if your platform doesn't support OpenGL.

To enable `wxGLCanvas` support under Windows, edit `include/wx/msw/setup.h`, set `wxUSE_GLCANVAS` to `1`, and compile with `USE_OPENGL=1` on the command line. You may also need to add `opengl32.lib` to the list of libraries your program is linked with. On Unix and Mac OS X, pass `--with-opengl` to the `configure` script to compile using OpenGL or Mesa.

If you're already an OpenGL programmer, using `wxGLCanvas` is very simple. You create a `wxGLCanvas` object within a frame or other container window, call `wxGLCanvas::SetCurrent` to direct regular OpenGL commands to the window, issue normal OpenGL commands, and then call `wxGLCanvas::SwapBuffers` to show the OpenGL buffer on the window.

The following paint handler shows the principles of rendering 3D graphics and draws a cube. The full sample can be compiled and run from `samples/opengl/cube` in your wxWidgets distribution.

```
void TestGLCanvas::OnPaint(wxPaintEvent& event)
{
    wxPaintDC dc(this);

    SetCurrent();

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-0.5f, 0.5f, -0.5f, 0.5f, 1.0f, 3.0f);
    glMatrixMode(GL_MODELVIEW);

    /* clear color and depth buffers */
    glClear(GL_COLOR_BUFFER_BIT ¦ GL_DEPTH_BUFFER_BIT);

    /* draw six faces of a cube */
    glBegin(GL_QUADS);
```

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 169

```
    glNormal3f( 0.0f, 0.0f, 1.0f);
    glVertex3f( 0.5f, 0.5f, 0.5f); glVertex3f(-0.5f, 0.5f, 0.5f);
    glVertex3f(-0.5f,-0.5f, 0.5f); glVertex3f( 0.5f,-0.5f, 0.5f);

    glNormal3f( 0.0f, 0.0f,-1.0f);
    glVertex3f(-0.5f,-0.5f,-0.5f); glVertex3f(-0.5f, 0.5f,-0.5f);
    glVertex3f( 0.5f, 0.5f,-0.5f); glVertex3f( 0.5f,-0.5f,-0.5f);

    glNormal3f( 0.0f, 1.0f, 0.0f);
    glVertex3f( 0.5f, 0.5f, 0.5f); glVertex3f( 0.5f, 0.5f,-0.5f);
    glVertex3f(-0.5f, 0.5f,-0.5f); glVertex3f(-0.5f, 0.5f, 0.5f);

    glNormal3f( 0.0f,-1.0f, 0.0f);
    glVertex3f(-0.5f,-0.5f,-0.5f); glVertex3f( 0.5f,-0.5f,-0.5f);
    glVertex3f( 0.5f,-0.5f, 0.5f); glVertex3f(-0.5f,-0.5f, 0.5f);

    glNormal3f( 1.0f, 0.0f, 0.0f);
    glVertex3f( 0.5f, 0.5f, 0.5f); glVertex3f( 0.5f,-0.5f, 0.5f);
    glVertex3f( 0.5f,-0.5f,-0.5f); glVertex3f( 0.5f, 0.5f,-0.5f);

    glNormal3f(-1.0f, 0.0f, 0.0f);
    glVertex3f(-0.5f,-0.5f,-0.5f); glVertex3f(-0.5f,-0.5f, 0.5f);
    glVertex3f(-0.5f, 0.5f, 0.5f); glVertex3f(-0.5f, 0.5f,-0.5f);
    glEnd();

    glFlush();
    SwapBuffers();
}
```

Figure 5-10 shows another OpenGL sample, a cute (if angular) penguin that can be rotated using the mouse. You can find this sample in `samples/opengl/penguin`.

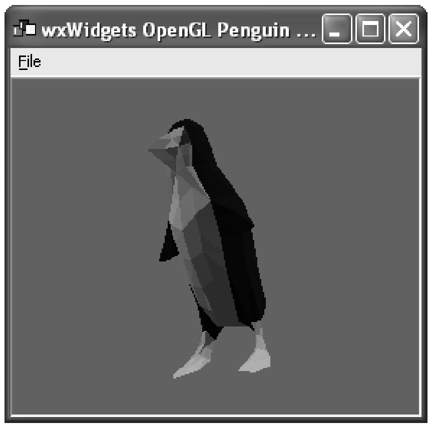


Figure 5-10 OpenGL "penguin" sample

Smart_Ch05f.qxd 6/10/05 11:17 AM Page 170

SUMMARY

In this chapter, you have learned how to draw on device contexts and use the wxWidgets printing framework, and you received a quick introduction to `wxGLCanvas`. You can look at the following source code in your wxWidgets distribution for examples of drawing and printing code:

- ☞ `samples/drawing`
- ☞ `samples/font`
- ☞ `samples/erase`
- ☞ `samples/image`
- ☞ `samples/scroll`
- ☞ `samples/printing`
- ☞ `src/html/htmprint.cpp`
- ☞ `demos/bombs`
- ☞ `demos/fractal`
- ☞ `demos/life`

For advanced 2D drawing applications, you might want to consider the wxArt2D library, which offers loading and saving of graphical objects using SVG files (Scalable Vector Graphics), flicker-free updating, gradients, vector paths, and more. See Appendix E, " Third-Party Tools for wxWidgets," for where to get wxArt2D.

Next, we'll look at how your application can respond to mouse, keyboard, and joystick input.