Глава 5

Рисование и печать

В этой главе вы познакомитесь с идеей контекста устройства, которая является обобщением поверхностей для рисования, таких как окно или страница принтера. Мы рассмотрим доступные классы контекстов устройств, а также множество различных инструментов для рисования, которые позволяют использовать шрифты, цвета и заливку. Далее будут описаны функции рисования на контексте устройств и использование wxWidgets для печати на принтере. В конце главы кратко рассмотрен класс `wxGLCanvas`, который используется для построения трехмерной графики в вашем окне с использованием библиотеки OpenGL.

5.1 Контекст устройства

Рисование в wxWidgets всегда производится на контексте устройства, который является классом-наследником от `wxDC`. В библиотеке нет такой вещи, как рисование непосредственно в окне. Вместо этого вы создаете контекст устройства для окна и рисуете на этом контексте. Также существуют отдельные контексты устройств для изображений и принтеров. Приятным следствием использования такой абстракции является то, что вы можете сделать один и тот же код рисования, и он будет работать на множестве различных контекстов устройств: просто параметризуете его с помощью `wxDC` и, если необходимо, передайте контексту правильный коэффициент масштабирования. Опишем основные свойства контекста устройства.

Контекст устройства имеет свою собственную систему координат с началом в левом верхнем углу. Позиция начала координат может быть изменена с помощью метода `SetDeviceOrigin`, в результате чего будет казаться, что контекст устройства сдвинут. Данный метод часто используется при рисовании на `wxScrolledWindow`. Вы также имеете возможность изменить ориентацию осей с помощью `SetAxisOrientation`. К примеру, можно установить направление оси `y` снизу наверх.

Существует различие между логическими (logical) и физическими (device) единицами измерения. Физические — это единицы измерения, специфичные для данного устройства. В частности для экрана таким единицами измерения являются пиксели. Для принтера размер его физических параметров определяется разрешением принтера и может быть получен с помощью методов `GetSize` (размер

страницы в физических единицах) или `GetSizeMM` (размер страницы в миллиметрах).

Режим отображения в контексте устройства определяет единицы измерения, используемые для преобразования логических единиц в физические. Заметим, что некоторые контексты устройств, в частности `wxPostScriptDC`, не поддерживают никаких режимов отображения кроме `wxMM_TEXT`. В таблице 5.1 представлены доступные режимы отображения.

Таблица 5.1: Режимы отображения

Идентификатор	Описание
`wxMM_TWIPS`	Каждая логическая единица имеет размер 1/20 точки (или 1/1440 дюйма).
`wxMM_POINTS`	Каждая логическая единица имеет размер одной точки (или 1/72 дюйма).
`wxMM_METRIC`	Каждая логическая единица имеет размер в 1 миллиметр.
`wxMM_LOMETRIC`	Каждая логическая единица имеет размер в 1/10 миллиметра.
`wxMM_TEXT`	Каждая логическая единица имеет размер в 1 пиксель. Это режим по умолчанию.

Вы можете дополнительно промасштабировать ваши логические единицы с помощью метода `SetUserScale`, в который передается соответствующий коэффициент. Например, в режиме `wxMM_TEXT` коэффициент `(1.0, 1.0)` делает логические и физические единицы одинаковыми. По умолчанию ипользуется режим отображения `wxMM_TEXT` и коэффициент `(1.0, 1.0)`.

Контекст устройства имеет область отсечения, координаты которой устанавливаются с помощью `SetClippingRegion` и очищаются с помощью `DestroyClippingRegion`. За границами отсечения графика не отрисовывается. Одним из полезных использований данного свойства является отрисовка строки, которая должна быть только внутри заданного прямоугольника, даже если строка превышает его размеры. Чтобы это сделать вам будет достаточно установить размер области отсечения равной прямоугольнику, написать текст и уничтожить область отсечения. В результате текст будет находится только внутри заданной области.

Как и у настоящих художников в начале необходимо выбрать инстумент для рисования. Любые операции рисования используют для обводки изображения текущее перо, а для заливки — текущую кисть. Текущий шрифт, вместе с внешним (foreground) и задним (background) фоном определяют вид отображаемого текста. Мы рассмотрим все эти инструменты позднее, а для начала изучим контесты устройств, доступные в библиотеке.

5.1.1 Доступные контексты устройств

Вот классы, которые вы можете использовать в своих программах:

- `wxClientDC`. Для рисования в клиентской области окна.
- `wxBufferedDC`. Замена для `wxClientDC` с двойной буферизацией.

- `wxWindowDC`. Для рисовния на клиентской и неклиентской (декорированной) области окна. Это очень редко используемый класс, который реализован не для всех платформ.
- `wxPaintDC`. Для рисования в клиентской области окна во время обработки сообщения о рисовании.
- `wxBufferedPaintDC`. Замена для `wxPaintDC` с двойной буферизацией.
- `wxScreenDC`. Для рисования на экране или копирования с него.
- `wxMemoryDC`. Для рисования на битовой карте или копирования с нее.
- `wxMetafileDC`. Для создания метафайла (в Windows и Mac OS X).
- `wxPrinterDC`. Для рисования на принтере.
- `wxPostScriptDC`. Для рисования в PostScript-файл или принтер.

Следующие разделы расскажут как правильно использовать данные контексты. Работа с принтерами подробно описана в разделе «Печать».

5.1.2 Рисование в окне с использованием wxClientDC

Используйте объект `wxClientDC` для рисования на клиентской области окна (но только не в обработчике сообщения о рисовании). Например, для реализации эскизного приложения вам может понадобится создать объект `wxClientDC` внутри обработчика сообщений от мыши. Данный объект также можно использовать внутри обработчика сообщения о закраске фона.

Вот фрагмент кода, демонстрирующего рисование в окне с использованием мыши:

```
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); // красное перо с шириной 1
        dc.SetPen(pen);
        dc.DrawPoint(event.GetPosition());
        dc.SetPen(wxNullPen);
    }
}
```

За примером более реалистичного кода для эскизного приложения обратитесь к Главе 19 «Технология документ/вид». Пример «Эскиз» использует сегменты линий вместо точек и поддерживает операцию отката. Приложение также хранит

все сегменты линий, поэтому приложение может перерисовывать свое окно. Изображение будет находится на холсте до получения следующего сообщения о перерисовке. Возможно вы также захотите использовать `CaptureMouse` и `ReleaseMouse`, чтобы перенаправлять все сообщения от мыши своему окну, пока нажата клавиша мыши.

Альтернативой использования `wxClientDC` является класс `wxBufferedDC`, который хранит результат вашего рисования в контексте устройства памяти и посылает изображение в окно за один раз при уничтожении данного контекста. В результате обновления происходят более гладко и подходят для случая, если вы не хотите, чтобы пользователь видел попиксельное обновление экрана. Использование данного класса почти ничем не отличается от использования `wxClientDC`. В целях эффективности вы можете передавать сохраненную битовую карту в конструктор класса, чтобы избежать ее создание каждый раз.

5.1.3 Очистка фона

Окно получает два вида сообщений о рисовании: `wxPaintEvent` для обновления основной части графики и `wxEraseEvent` для рисования фона. Если вы будете перехватывать только сообщение `wxPaintEvent`, то стандартный обработчик для `wxEraseEvent` каждый раз будет очищает фон цветом, который задан с помощью `wxWindow::SetBackgroundColour`.

Хотя это выглядит немного обескураживающе, но разделение рисования фона и изображения дает больше возможностей на платформах, которые следуют этой модели, например, в Windows. Предположим вы решили рисовать текстурированный фон в окне. Если вы будете накладывать текстуру в обработчике `OnPaint`, то будете видеть мерцание, так как рисование фона идет до рисования текстур. Чтобы избежать этого необходимо сделать обработчик события `wxEraseEvent` пустым. С другой стороны вы можете в обработчике `wxEraseEvent` делать наложение текстур, а рисование делать в `OnPaint` (однако в таком случае сложно сделать буферизацию, о которой будет рассказано в следующем разделе).

На некоторых платформах перехватывать `wxEraseEvent` не достаточно, чтобы подавить стандартную очистку фона. Самый безопасный метод этого достичь — это вызвать метод `wxWindow::SetBackgroundStyle` и передать ему `wxBG_STYLE_CUSTOM`. Данный флаг говорит о том, что приложение самостоятельно будет рисовать фон.

Если вы решили реализовать обработчик заливки фона, то в обработчике сначала вызовите `wxEraseEvent::GetDC` и используйте возвращенное значение, если оно существует. Если это значение равно `NULL`, то вы можете использовать контекст `wxClientDC`. Это позволит корректно работать реализациям wxWidgets, которые не передают контекст устройства в обработчик сообщения. Данная методика демонстрируется в примере, который использует рисунок в качестве фона:

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

Как и в случае сообщения о рисовании контекст устройства может содержать область отсечения, которую необходимо отрисовать. Для этого достаточно получить объект с помощью `wxEraseEvent::GetDC`.

5.1.4 Рисование в окне с помощью wxPaintDC

Если вы переопределяете обработчик события о рисовании, то вам всегда необходимо создавать внутри объект `wxPaintDC`, даже если он вам не нужен. Создание этого объекта говорит wxWidgets, что поврежденная область окна уже перерисована, а поэтому оконная система не должна посылать сообщение о рисовании до бесконечности. В сообщении о рисовании вы можете вызвать метод `wxWindow::GetUpdateRegion`, чтобы получить поврежденную область или `wxWindow::IsExposed`, чтобы определить находится ли данная точка или прямоугольник в поврежденном регионе. Если это возможно, то просто перерисуйте данный регион. Контекст устройства автоматически передает необходимую область отсечения внутри события о рисовании, но вы можете ускорить работу перерисовав только то, что необходимо.

Событие о перерисовке генерируется в случае, когда действия пользователя приводят к тому, что окну необходимо перерисовать часть своей области, но это событие также может быть следствием вызова метода `wxWindow::Refresh` или `wxWindow::RefreshRect`. Если вы точно знаете какую область необходимо перерисовать, то можете обновить только эту область, чем максимально снизите возможное мерцание. Главной проблемой при обновлении окна является то, что вы не можете гарантировать момент, когда оно на самом деле будет обновлено. Если вам необходимо форсировать обновление окна (например во время продолжительных вычислений), то вы можете вызвать метод `wxWindow::Update` после предварительного вызова `Refresh` или `RefreshRect`.

Следующий код рисует в центре окна красный прямоугольник с черной обводкой. Обновление изображения делается только в случае необходимости:

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

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

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

    // Берем размеры окна
    wxSize sz = GetClientSize();

    // Размеры нашего прямоугольника
    wxCoord w = 100, h = 50;

    // Центрируем прямоугольник в окне, но никогда не рисуем
    // в отрицательных координатах
    int x = wxMax(0, (sz.xw)/2);
    int y = wxMax(0, (sz.yh)/2);

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

    // Для эффективности не перерисовываем, если прямоугольник
    // не поврежден
    if (IsExposed(rectToDraw))
        DrawRectangle(rectToDraw);
}
```

Заметим, что по умолчанию при изменении размера окна в регион для обновления добавляются только новые области. Используйте оконный стиль `wxFULL_REPAINT_ON_RESIZE`, чтобы изменение размера требовало полной перерисовки окна. В примере мы вынуждены использовать данный стиль, так как изменение размеров окна изменяет расположение графики, поэтому необходимо убедиться, что слева не находится несуществующих областей картинки.

`wxBufferedPaintDC` — это версия `wxPaintDC` с буферизацией. Просто замените `wxPaintDC` на `wxBufferedPaintDC` в вашем обработчике сообщений, и графика будет рисоваться на битовой карте, а в конце быстро скопирована в окно, тем самым уменьшая мерцание.

Как упоминалось в предыдущей теме существует еще один способ сделать прорисовку более гладкой (в частности при изменении размеров окна). Для этого необходимо отрисовывать фон в обработчике события о рисовании, а не в обработчике события о перерисовке фона. В таком случае все рисование можно делать в буферизированном обработчике события о рисовании, а потому вы не увидите очистку фона перед вызовом события о рисовании. Просто добавьте пустой обработчик события о перерисовке фона, вызовете метод `SetBackgroundStyle` с флагом `wxBG_STYLE_CUSTOM`, чтобы сказать некоторым системам не очищать фон принудительно. В прокручивающихся окнах, где точка начала координат смещается до текущей прокручивающейся позиции необходимо самостоятельно вычислять

положение клиентской области окна для текущего начала координат. Следующий кусок кода иллюстрирует как можно достичь гладкого рисования и прокрутку для наследников от wxScrolledWindow:

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

BEGIN_EVENT_TABLE(MyCustomCtrl, wxScrolledWindow)
    EVT_PAINT(MyCustomCtrl::OnPaint)
    EVT_ERASE_BACKGROUND(MyCustomCtrl::OnEraseBackground)
END_EVENT_TABLE()

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

    // Смещаем текущее начало координат, а поэтому нам не надо
    // самостоятельно вычислять это значение
    PrepareDC(dc);

    // Рисуем фон
    PaintBackground(dc);

    // Рисуем графику
    ...
}

/// Рисуем фон
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());

    // Необходимо сдвинуть клиентский прямоугольник, чтобы рисовать
    // на правильных координатах
    CalcUnscrolledPosition(windowRect.x, windowRect.y,
                           & windowRect.x, & windowRect.y);
    dc.DrawRectangle(windowRect);
}
```

```
// Пустая реализация, чтобы избежать ненужного мерцания
void MyCustomCtrl::OnEraseBackground(wxEraseEvent& event)
{
}
```

Чтобы повысить эффективность использования `wxBufferedPaintDC`, можно создать битовую карту с достаточными размерами для любого размера окна (например, для этого можно взять размеры экрана) и подставить данную битовую карту в качестве второго аргумента конструктора `wxBufferedPaintDC`. В результате контекст устройства не будет создавать и уничтожать битовую карту на каждое событие о рисовании.

Область, которую `wxBufferedPaintDC` копирует из своего буфера обычно имеет размер клиентской области окна (той части, которую видит пользователь). Контекст рисования, который создается данным классом, не преобразуется оконным `PrepareDC`, а поэтому не соответствует оконному положению бегунков. Однако, вы можете указать, чтобы контекст для рисования и ваш буферизированный контекст использовали одинаковые преобразования, передав `wxBUFFER_VIRTUAL_AREA` в конструктор `wxBufferedPaintDC`, вместо стандартного `wxBUFFER_CLIENT_AREA`. Будет вызван метод `PrepareDC` вашего окна, поэтому преобразования обоих контекстов будут совпадать. Для такого случая необходимо предоставить битовую карту, которая имеет тот же размер, что и виртуальная область в вашем прокручивающемся окне. Такое решение не эффективно и должно по возможности избегаться. Заметим, что на момент написания, использование буферизации `wxBUFFER_CLIENT_AREA` не работало с масштабированием (`SetUserScale`).

Пример использования класса `wxBufferedPaintDC` вы можете увидеть в реализации `wxThumbnailCtrl` в папке examples/chap12/thumbnail на прилагаемом CD-ROM.

5.1.5 Рисование на битовых картах с помощью wxMemoryDC

Контекст устройства в памяти всегда имеет связанную с ним битовую карту, поэтому рисование в такой контекст означает рисование на связанной битовой карте. Контекст в памяти использовать достаточно просто. Сначала с помощью конструктора по умолчанию создается объект класса `wxMemoryDC`, далее вызывается метод `SelectObject` для привязки битовой карты к контексту. После использования контекста памяти необходимо вызвать `SelectObject` с параметром `wxNullBitmap`, чтобы удалить связывание.

Следующий пример создает битовую карту и рисует на ней красный прямоугольник:

```
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;
}
```

Также можно скопировать область из контекста в памяти в другой контекст с помощью метода `Blit`, который описывается далее в главе.

5.1.6 Использование wxMetafileDC для создания метафайлов

Класс `wxMetafileDC` доступен на платформах Windows и Mac OS X, где он служит прослойкой для создания метафайлов Windows и Mac PICT соответственно. Данный класс предоставляет возможность рисования в объект `wxMetafile`, который содержит список инструкций для слоя рисования, которые могут интерпретироваться самим приложением или отрисоваться в контексте устройства с помощью `wxMetafile::Play`.

5.1.7 Рисование на экране при помощи wxScreenDC

Класс `wxScreenDC` используется, чтобы рисовать в произвольной области экрана. Это иногда полезно, если вы хотите предусмотреть некоторую визуальную реакцию на операцию перетаскивания. В целях оптимизации можно ограничить область на экране, определив регион (чаще всего равным размерам самого окна). Кроме того данный класс можно использовать, чтобы захватить изображение с экрана (сделать снимок).

Далее приведен пример кода, который делает снимок текущего экрана и возвращает соответствующую битовую карту:

```
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;
}
```

5.1.8 Печать с использованием wxPrinterDC и wxPostScriptDC

`wxPrinterDC` является поверхностью для печати. В системах Windows и Mac этот класс напрямую взаимодействует с соответствующей подсистемой печати. В

Unix-системах, где отсутствует стандартная модель печати, обычно используется `wxPostScriptDC`. Если в системе доступна система печати GNOME, то лучше использовать именно ее (смотрите далее в разделе «Печать в Unix с использованием GTK+»).

Существует несколько способов создания объекта `wxPrinterDC`. Например, можно передать объект `wxPrintData`, в котором предварительно можно установить тип бумаги, ориентацию, число копий и так далее. Самый простой путь — показать `wxPrintDialog`, а потом вызвать `wxPrintDialog::GetPrintDC` для получения подходящего контекста `wxPrinterDC` с установками, выбранными пользователем. На более высоком уровне можно создать наследника от `wxPrintout` и определить в нем поведение при печати и при предпросмотре, а когда необходимо передать ваш класс экземпляру объекта `wxPrinter` (о печати более подробно будет рассказано в следующих разделах).

Если необходимо печатать по большей части только текст, то ознакомьтесь с классом `wxHtmlEasyPrinting`, который позволит вам избежать использования `wxPrinterDC` или `wxPrintout`: просто создайте корректный HTML-файл (используя поддерживаемое wxWidgets подмножество синтаксиса HTML) и создайте объект `wxHtmlEasyPrinting` для его печати или предпросмотра. Данный класс сэкономит вам дни (или даже недели) программирования, которое иначе бы потребовалось для правильного оформления вашего текста, таблиц и изображений. Обратитесь к Главе 12 «Сложные классы окон» за дополнительной информацией.

Класс `wxPostScriptDC` — это контекст устройства, предназначенный для создания PostScript-файлов, которые можно посылать непосредственно принтеру. Хотя данный класс и создан по большей части для Unix-систем, но его также можно использовать и для других систем, когда необходимо создать файл в формате PostScript, но у вас нет гарантии, что установлен PostScript-драйвер.

Можно создать `wxPostScriptDC`, передав ему объект `wxPrintData` или имя файла для сохранения. Дополнительный булевый параметр говорит о том, показывать ли диалог печати пользователю. Например,

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

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

if (dc.Ok())
{
    // Указываем местоположение файлов AFM
    dc.GetPrintData().SetFontMetricPath(wxGetApp().GetFontPath());

    // Устанавливаем рарешение в точках на дюйм (по умолчанию 720)
    dc.SetResolution(1440);

    // Рисуем на контексте устройства
    ...
}
```

Одно из неудобств класса `wxPostScriptDC` состоит в том, что он не может прямо возвратить информацию о размере символов из `GetTextExtent`. Чтобы

это стало возможным вам необходимо поставлять специальные AFM-файлы (Adobe Font Metric) вместе с вашим приложением, а также использовать метод `wxPrintData::SetFontMetricPath`, чтобы библиотека wxWidgets могла их найти, как это показано в нашем примере. AFM файлы для некоторых шрифтов вы можете скачать по адресу ftp://biolpc22.york.ac.uk/pub/support/gs_afm.tar.gz.

5.2 Инструменты для рисования

Код для рисования в wxWidgets работает как очень быстрый художник: мгновенно выбирает цвет и инструмент, рисует маленькую часть картины, далее выбирает другой инструмент и рисует другую часть картины и так далее. В данном разделе описываются инструменты для рисования: классы `wxColour`, `wxPen`, `wxBrush`, `wxFont` и `wxPalette`. В Главе 13 «Структуры данных» вы также сможете найти полезную информацию по различным вспомогательным классам для рисования, таким как `wxRect`, `wxRegion`, `wxPoint` и `wxSize`.

Обратите внимание, что все эти классы используют достаточно эффективный механизм «подсчета ссылок», который позволяет в большинстве случаев копировать только указатель, а не куски памяти. Таким образом чаще всего вы можете создавать цвета, обводки, кисти и шрифты в стеке, не беспокоясь о потере скорости. Однако, если в вашем приложении есть проблемы со скоростью, то в некоторых случаях можно увеличить скорость работы, сделав такие объекты полями классов.

5.2.1 wxColour

Класс `wxColour` необходимо использовать, чтобы определить различные параметры цвета во время рисования (из-за того, что проект wxWidgets родился в Эденбурге его API использует британское написание слова «цвет», однако позже для соответствия более распространенному американскому языку в wxWidgets был определен `wxColor` как синоним для `wxColour`).

Вы можете установить цвет фона и символов текста для контекста устройства, используя методы `SetTextForeground` и `SetTextBackground`. Также можно передать параметр класса `wxColour` при создании кистей и перьев.

Объект класса `wxColour` можно создать множеством различных способов. Вы можете передать интенсивность красного, зеленого и синего (значение от 0 до 255), или стандартную строку с цветом (такую как WHITE или CYAN). Также можно создать цвет из другого объекта `wxColour`. Кроме того, можно использовать предопределенные объекты, которые являются указателями на цветовые объекты: `wxBLACK`, `wxWHITE`, `wxRED`, `wxBLUE`, `wxGREEN`, `wxCYAN` и `wxLIGHT_GREY`. Существует объект `wxNullColour`, который по сути является неинициализированным цветом для которого метод `Ok` всегда возвращает `false`.

Используя класс `wxSystemSettings` можно получить некоторые стандартные системные цвета, такие цвет обычной 3D-поверхности, обычный цвет фона окна, цвет текста меню и так далее. Обратитесь к документации по `wxSystemSettings::GetColour` за списком возможных идентификаторов, которые можно передать классу.

Вот различные примеры создания объектов `wxColour`:

```
wxColour color(0, 255, 0);  // зеленый
wxColour color(wxT("RED")); // красный

// Следующий цвет используется для 3D-элементов и панелей
wxColour color(wxSystemSettings::GetColour(wxSYS_COLOUR_3DFACE));
```

Также можно использовать указатель на `wxTheColourDatabase`, чтобы добавить новый цвет, найти объект класса `wxColour` по заданному имени, или найти имя, сопоставленное данному цвету:

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

Список доступных стандартных цветов указан в таблице 5.2.

Таблица 5.2: Список цветов, доступных по имени

Идентификатор	Имя цвета
aquamarine	аквамариновый
black	черный
blue	синий
blue violet	фиолетово-синий
brown	коричневый
cadet blue	серо-синий
coral	коралловый
cornflower blue	васильковый синий
cyan	голубой
dark gray	темно-серый
dark green	темно-зеленый
dark olive green	темный оливковый зеленый
dark	темный
orchid	орхидея
dark slate blue	темно-синий сланцевый
dark slate gray dark turquoise	темно-синевато-серый бирюзовый
dim gray	тускло-серый
firebrick	цвет кирпича
forest	лесной
green	зеленый
gold	золотой
goldenrod	золотарник
gray	серый
green	зеленый
green yellow	зелено-желтый
indian red	индийская краснота
khaki	хаки
light blue	светло-голубой
light gray	светло-серый

Таблица 5.2: Список цветов, доступных по имени (продолжение)

Идентификатор	Имя цвета
light steel blue	светло-синий стальной
lime green	лимонно-зеленый
magenta	фуксин
maroon	марон
medium aquamarine	средний аквамарин
medium blue	средний синий
medium forest green	средний темно-зеленый
medium goldenrod	средний золотарник
medium orchid	средняя орхидея
medium sea green	средний цвет морской волны
medium slate blue	средний синий сланецевый
medium spring green	средний весенний зеленый
medium turquoise	средний бирюзовый
medium violet red	средний фиолетово-красный
midnight blue	синяя полночь
navy	морской
orange	оранжевый
orange red	оранжево-красный
orchid	орхидея
pale green	бледно-зеленый
pink	розовый
plum	слива
purple	фиолетовый
red	красный
salmon	лосось
sea green	цвет морской волны
sienna	сиена
sky blue	лазурный
slate blue	синий сланецевый
spring green	весенний зеленый
steel blue	синий стальной
tan	коричневый
thistle	чертополох
turquoise	бирюзовый
violet	фиолетовый
violet red	фиолетово-красный
wheat	пшеница
white	белый
yellow	желтый
yellow green	желто-зеленый

5.2.2 wxPen

Чтобы определить текущие параметры обводки (пера, карандаша) для контекста устройства необходимо передать объект класса `wxPen` в метод `SetPen`. Обводка определяет цвет, толщину и стиль. Класс `wxPen` достаточно «легкий», поэтому можно определять его экземпляры в стеке вместо того, чтобы постоянно хранить его.

Все возможные стили для обводок описаны в таблице 5.3. Стили решетки (hatch) и гравировки (stipple) в данный момент не поддерживаются в GTK+.

Таблица 5.3: Стили wxPen

Стили	Пример	Описание
`wxSOLID`		Линия рисуется сплошной.
`wxTRANSPARENT`		Используется, когда делать обводку не надо.
`wxDOT`		Линии рисуются точками.
`wxLONG_DASH`		Линии рисуются длинными отрезками.
`wxSHORT_DASH`		Линии рисуются короткими отрезками. В системе Windows аналогичен `wxLONG_DASH`.
`wxDOT_DASH`		Линии рисуются точками с отрезками.
`wxSTIPPLE`		Использует изображение, которое передается в качестве первого аргумента конструктора.
`wxUSER_DASH`		Использует пользовательскую обводку. Обратитесь к документации за большей информацией.
`wxBDIAGONAL_HATCH`		Рисует как обратно-диагональную решетку.
`wxCROSSDIAG_HATCH`		Рисует как крестовую решетку.
`wxFDIAGONAL_HATCH`		Рисует как прямо-диагональную решетку.
`wxCROSS_HATCH`		Рисует как прямую крестовую решетку.
`wxHORIZONTAL_HATCH`		Рисует как горизонтальную решетку.
`wxVERTICAL_HATCH`		Рисует как вертикальную решетку.

Метод `SetCap` позволяет изменить способ начертания концов обводки: с установленным `wxCAP_ROUND` (по умолчанию) концы закругляются, с `wxCAP_PROJECTING` концы рисуются квадратными, а с `wxCAP_BUTT` концы рисуются

квадратными и не один из них не отбрасывается.

С помощью метода SetJoin можно определить способ соединения линий. По умолчанию используется стиль wxJOIN_ROUND, который рисует углы скругленными. Другие возможные значения — wxJOIN_BEVEL и wxJOIN_MITER.

Существует несколько предопределенных обводок, которые можно использовать: wxRED_PEN, wxCYAN_PEN, wxGREEN_PEN, wxBLACK_PEN, wxWHITE_PEN, wxTRANSPARENT_PEN, wxBLACK_DASHED_PEN, wxGREY_PEN, wxMEDIUM_GREY_PEN и wxLIGHT_GREY_PEN. Эти константы являются указателями, а поэтому их надо разыменовывать при передаче в SetPen. Существует также объект wxNullPen (это именно объект, а не указатель на него), который обозначает неинициализированную обводку. Его можно использовать, чтобы сбросить текущую обводку в контексте устройства.

Вот несколько примеров создания обводок:

```
// Сплошная красная обводка
wxPen pen(wxColour(255, 0, 0), 1, wxSOLID);
wxPen pen(wxT("RED"), 1, wxSOLID);
wxPen pen = (*wxRED_PEN);
wxPen pen(*wxRED_PEN);
```

В последних двух строчках примера используется подсчет ссылок, поэтому внутренние данные обводок указывают на данные wxRED_PEN. Подсчет ссылок используется для всех объектов рисования, что делает операции присваивания и копирования очень быстрыми, но это также означает, что в некоторых случаях изменение одного объекта приведет к изменению свойств другого.

Одним из способов сократить число созданий/удалений классов обводок без их хранения внутри ваших классов заключается в использовании глобального указателя wxThePenList для создания и хранения необходимых вам обводок. Например,

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

В таком случае обводка будет хранится в wxThePenList и освобождаться при закрытии программы. Однако необходимо быть острожными и бесконтрольно не использовать данный метод, так как это может привести к излишнему расходу памяти на хранение ненужных обводок. Кроме того, необходимо учитывать нюансы механизма подсчета ссылок, которые недавно упоминались. Метод RemovePen позволяет удалить обводку из списка, без освобождения памяти.

5.2.3 wxBrush

Текущая кисть, задаваемая с помощью SetBrush, определяет цвет и стиль закраски. Существует возможность задать цвет фона контекста устройства с помощью объекта wxBrush. Также как и в случае с wxPen, wxBrush обладает низкими накладными расходами на создание, а поэтому может создаваться в стеке.

В конструктор кисти передается цвет и стиль. Стиль может быть одним из приведенных в таблице 5.4.

Таблица 5.4: Стили wxBrush

Стили	Пример	Описание
wxSOLID		Используется сплошная кисть.
wxTRANSPARENT		Данный стиль используется, если необходимо отключить закраску.
wxBDIAGONAL_HATCH		Рисует как обратно-диагональную решетку.
wxCROSSDIAG_HATCH		Рисует как крестовую решетку.
wxFDIAGONAL_HATCH		Рисует как прямо-диагональную решетку.
wxCROSS_HATCH		Рисует как прямую крестовую решетку.
wxHORIZONTAL_HATCH		Рисует как горизонтальную решетку.
wxVERTICAL_HATCH		Рисует как вертикальную решетку.
wxSTIPPLE		Используется битовая карта, которая передается в качестве первого аргумента конструктора.

Существует несколько предопределенных кистей, которые вы можете использовать: `wxBLUE_BRUSH`, `wxGREEN_BRUSH`, `wxWHITE_BRUSH`, `wxBLACK_BRUSH`, `wxGREY_BRUSH`, `wxMEDIUM_GREY_BRUSH`, `wxLIGHT_GREY_BRUSH`, `wxTRANSPARENT_BRUSH`, `wxCYAN_BRUSH` и `wxRED_BRUSH`. Все эти константы являются указателями, а поэтому их необходимо разыменовывать при передаче в `SetBrush`. Существует также объект `wxNullBrush` (это именно объект, а не указатель на него), который обозначает неинициализированную кисть. Его можно использовать, чтобы сбросить текущую кисть в контексте устройства.

Вот несколько примеров создания кистей:

```
// Сплошная красная кисть
wxBrush brush(wxColour(255, 0, 0), wxSOLID);
wxBrush brush(wxT("RED"), wxSOLID);
wxBrush brush = (*wxRED_BRUSH); // дешевая операция
wxBrush brush(*wxRED_BRUSH);
```

Так же, как и в случае с `wxPen`, для `wxBrush` существует аналогичный список `wxTheBrushList`, который можно использовать для кеширования кистей:

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

Используйте его с осторожностью, чтобы избежать излишнего распространения кистей, учитывая при этом возможные эффекты механизма подсчета ссылок. Удалить кисть из списка без освобождения из памяти возможно с помощью `RemoveBrush`.

5.2.4 wxFont

Объекты данного класса используются, чтобы задать каким шрифтом будет выводится текст на контекст устройства. Шрифт обладает следующими свойствами:

Размер шрифта определяет максимальную высоту текста в точках (1/72 дюйма). wxWidgets будет выбирать наиболее подходящий к этой величине шрифт, если платформа не поддерживает масштабирование шрифтов.

Семейство шрифтов выбирается из небольшого количества возможных семейств, которые перечислены в таблице 5.5. Определение семейства вместо явного указания имени шрифта более переносимо, так как нельзя гарантировать, что некоторый шрифт будет доступен на всех платформах.

Таблица 5.5: Идентификаторы семейств шрифтов

Идентификатор	Пример	Описание
`wxFONTFAMILY_SWISS`	ABCDEFGabcdefg12345	Рубленый шрифт. Обычно Helvetica или Arial в зависимости от платформы.
`wxFONTFAMILY_ROMAN`	ABCDEFGabcdefg12345	Шрифт с засечками.
`wxFONTFAMILY_SCRIPT`	ABCDEFGabcdefg12345	Рукописный шрифт.
`wxFONTFAMILY_MODERN`	ABCDEFGabcdefg12345	Шрифт фиксированной ширины, обычно Courier.
`wxFONTFAMILY_DECORATIVE`	ABCDEFGabcdefg12345	Декоративный шрифт.
`wxFONTFAMILY_DEFAULT`		Выбирается семейство по умолчанию.

Стиль может быть `wxNORMAL` (нормальный), `wxSLANT` (широкий) или `wxITALIC` (наклонный). Стиль `wxSLANT` доступен не для всех платформ и шрифтов.

Вес может быть `wxNORMAL` (нормальный), `wxLIGHT` (легкий) или `wxBOLD` (жирный).

Подчеркивание у шрифта может быть включено (`true`) или выключено (`false`).

Имя шрифта является необязательным параметром. Если данный параметр пуст, то имя выбирается в соответствии с назначенным семейством.

Дополнительный параметр *кодировка* устанавливает связь между кодом символа, используемым в программе, и буквами, отображаемыми на контексте устройства. Обратитесь к Главе 16 «Написание локализованных приложений» за дополнительной информацией по этому вопросу.

Чтобы создать шрифт необходимо воспользовавшись конструктором по умолчанию или определить его свойства, перечисленные в таблице 5.5.

Существует несколько определяемых библиотекой шрифтов, которые вы можете использовать в своих программах: `wxNORMAL_FONT` (обычный шрифт), `wxSMALL_FONT` (маленький шрифт), `wxITALIC_FONT` (наклонный шрифт) и `wxSWISS_FONT`. Все они имеют размер стандартный для шрифтов системы (`wxSYS_DEFAULT_GUI_FONT`), кроме `wxSMALL_FONT`, который на два точки меньше. Вы также можете использовать `wxSystemSettings::GetFont`, чтобы получить стандартный шрифт.

Чтобы использовать объект класса `wxFont` необходимо передать его в метод `wxDC::SetFont`, перед операциями с текстом, в частности `DrawText` и `GetTextExtent`.

Приведем пример создания шрифтов:

```
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));
```

Для объектов wxFont есть соответствующий список (wxTheFontList), который вы можете использовать, чтобы найти же созданный шрифт или создать новый:

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

Как и со списками обводок и кистей используйте данный список с осторожностью, так как шрифты должны уничтожаться только при выходе из приложения. Вы можете удалить шрифт из списка без его удаления из памяти с помощью метода `RemoveFont`.

Некоторый примеры работы с текстом и шрифтами будут приведены далее в главе. Также рекомендуем вам ознакомится с примером работы со шрифтами в samples/font (см. рис. 5.1). Данное приложение позволит вам посмотреть как будет выглядеть текст, написанный выбранным шрифтом и размером.

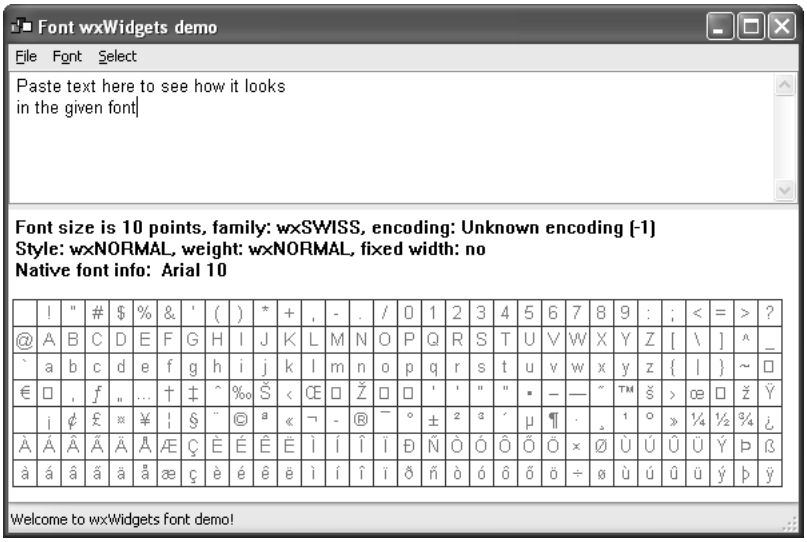

Рис. 5.1: Демонстрация работы со шрифтами

5.2.5 wxPalette

Палитра является таблицей (чаще всего размером в 256 элементов), которая отображает индекс в некоторую комбинацию красного, зеленого и синего цветов. Обычно палитры используются для экранов с поддержкой ограниченного числа цветов, которые необходимо разделять также и с другими приложениями. Поэтому, определяя палитру для клиентского контекста устройства, необходимо учитывать также потребности других приложений. Также палитры используются для отображения цветов битовой карты с маленькой глубиной (low-depth) на все доступные цвета, поэтому с объектом класса `wxBitmap` можно ассоциировать палитру `wxPalette`.

Из-за того, что компьютеры сейчас, как правило, имеют полноцветные мониторы, то палитры используются достаточно редко. RGB-цвет, определенный приложением отображается в ближайший цвет монитора без использования палитр.

Для создания объекта `wxPalette` передается размер и три массива (`unsigned char*`) для обозначения значений каждой компоненты цвета (красный, зеленый, синий). С помощью `GetColoursCount` можно получить текущее число цветов в палитре. Для получения значения красной, зеленой или синей компоненты цвета для некоторого индекса используйте `GetrGB`, а если необходимо по заданным значениям компонент найти ближайший цвет в палитре — `GetPixel`.

Установить палитру на клиенте, окне или контексте в памяти можно с помощью `wxDC::SetPalette`. Например, вы можете установить палитру, полученную из `wxBitmap` с маленькой глубиной перед ее рисованием, чтобы система знала как отображать индексные значения в цвета устройства. При использовании функций рисования, которые используют `wxColour`, на контексте устройства с установленной палитрой увета будут автоматически отображены системой на некоторые цвета из палитры, поэтому выбирайте палитру наиболее похожую на используемые в рисунке цвета.

Еще одним способом использования `wxPalette` является получение цветов из изображений `wxImage` или `wxBitmap` с маленькой глубиной, которые загружаются из файлов, таких как GIF-файлы. Если при зарузке вы обнаружите ассоциированный объект класса `wxPalette`, то получите простой способ получения всех уникальных цветов в изображении, даже если оно было преобразовано в RGB-представление. Аналогично, вы можете создать и ассоциировать палитру с `wxImage`, который будет сохранен в формате с ограничением цвета. Например, следующий фрагмент загружает PNG-файл и сохраняет его в 8-битный BMP-файл:

```
// Загружаем изображение в формате PNG
wxImage image(wxT("image.png"), wxBITMAP_TYPE_PNG);

// Создаем палитру
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);

// Устанавливаем глубину изображения для палитры и BMP
image.SetPalette(palette);
image.SetOption(wxIMAGE_OPTION_BMP_FORMAT, wxBMP_8BPP_PALETTE);

// Сохраняем изображение в файл
image.SaveFile(wxT("image.bmp"), wxBITMAP_TYPE_BMP);
```

Более реалистичный код должен «дискретизировать» изображение, чтобы сократить число цветов. В разделе «Уменьшение цветности» Главы 10 «Работа с

изображениями» рассказывается как использовать класс `wxQuantize`, чтобы сделать это.

В библиотеке wxWidgets также определен пустая палитра с именем `wxNullPalette`.

5.3 Методы рисования на контексте устройства

В данном разделе подробно рассказывается как происходит рисование на контексте устройства. Основные методы перечислены в таблице 5.6, большинство из которых будут подробно описаны далее. Дополнительная информация также доступна в руководстве по библиотеке.

Таблица 5.6: Методы контекста устройства

Имя	Описание
`Blit`	Производит копирование из одного контекста устройства в другой. Вы можете определить какую часть оригинального изображения копировать, использовать специальную логическую функцию копирования, а также использовать копируемое изображение в качестве маски (если источником является контекст устройства в памяти).
`Clear`	Заливает контекст устройства текущей фоновой кистью.

`SetClippingRegion`
`DestroyClippingRegion`
`GetClippingBox`
	Устанавливает и уничтожает область отсечения, которая ограничивает рисование некоторыми границами. Область отсечения может быть задана как прямоугольник или `wxRegion`. Используйте метод `GetClippingBox`, чтобы получить прямоугольник, вмещающий текущую область отсечения.

`DrawArc` `DrawEllipticArc`	Рисует дугу и элиптическую дугу, используя текущую обводку и кисть.
`DrawBitmap`	Рисует изображение `wxBitmap` или `wxIcon` в указанном месте.
`DrawIcon`	Рисует изображение, которое может иметь маску, определяющую прозрачность.
`DrawCircle`	Рисует окружность, используя текущую обводку и кисть.
`DrawEllipse`	Рисует эллипс, используя текущую обводку и кисть.
`DrawLine` `DrawLines`	Рисует линию или несколько линий, используя текущую обводку. Последняя точка линии не рисуется.
`DrawPoint`	Рисует точку, используя текущую обводку.

Таблица 5.6: Методы контекста устройства (продолжение)

Имя	Описание
`DrawPolygon` `DrawPolyPolygon`	`DrawPolygon` рисует закрашенный многоугольник, который задается списком точек или указателей на них. Дополнительно можно указать смещение координат. wxWidgets автоматически соединяет линией первую и последнюю точку. `DrawPolyPolygon` позволяет нарисовать несколько многоугольников за один раз, что является более быстрой операцией на некоторых платформах.
`DrawRectangle` `DrawRoundedRectangle`	Рисует прямоугольник или закругленный прямоугольник, используя текущую обводку и кисть.
`DrawText` `DrawRotatedText`	Рисует строку текста или повернутую строку на определенном месте, используя текущий шрифт, цвет текста и фона.
`DrawSpline`	Рисует сплайн между заданными контрольными точками, используя текущую обводку.
`FloodFill`	Заливает контекст устройства, начиная с заданной точки и используя текущий цвет кисти.
`Ok`	Возвращает `true`, если контекст устройства готов для использования.
`SetBackground` `GetBackground`	Установить и получить текущую фоновую кисть, которая используется в методе `Clear` и в функциях, использующих сложные логические функции. По умолчанию используется кисть `wxTRANSPARENT_BRUSH`.
`SetBackgroundMode` `GetBackgroundMode`	Устанавить и получить текущий режим фона, который используется при написании текста: `wxSOLID` или `wxTRANSPARENT`. По умолчанию используется режим `wxTRANSPARENT`, который означает, что фон под выводимым текстом не изменяется.
`SetBrush` `GetBrush`	Установить и получить кисть, используемую для заполнения фигур в операциях рисования. Начальное значение для этой кисти не задано.
`SetPen` `GetPen`	Установить и получить обводку, используемую при рисовании фигур. Начальное значение для обводки не задано.
`SetFont` `GetFont`	Установить и получить шрифт, используемый в `DrawText`, `DrawRotatedText` и `GetTextExtent`. Начальное значение для шрифта не задано.
`SetPalette` `GetPalette`	Установить и получить объект `wxPalette`, используемый для отображения целочисленных индексов в RGB-цвет.

Таблица 5.6: Методы контекста устройства (продолжение)

Имя	Описание

`SetTextForeground`
`GetTextForeground`
`SetTextBackground`
`GetTextBackground`
	Установить и получить цвет фона и текста, используемых при выводе символов. По умолчанию используется черный и белый цвет соответственно.

`SetLogicalFunction` `GetLogicalFunction`	Логическая функция, определяющая как исходный пиксель из обводки, кисти или из контекста устройства (при использовании операции `Blit`) комбинируется с пикселем в точке назначения в текущем контексте устройства. По умолчанию используется значение `wxCOPY`, которое означает, что старый пиксель зарисовывается текущим цветом.
`GetPixel`	Возвратить цвет указанной точки. Этот метод не реализован для `wxPostScriptDC` и `wxMetafileDC`.
`GetTextExtent` `GetPartialTextExtents`	Возвращает метрики для заданного текста.
`GetSize` `GetSizeMM`	Возвращает размеры контекста в единицах измерения контекста устройства или миллиметрах.
`StartDoc` `EndDoc`	Начинает и заканчивает печать документа. Данный метод применим только для контекста устройства принтера. При вызове `StartDoc` появляется сообщение, что документ печатается, а при вызове `EndDoc` сообщение скрывается.
`StartPage` `EndPage`	Начинает и заканчивает печать страницы. Данный метод применим только для контекста устройства принтера.

`DeviceToLogicalX`
`DeviceToLogicalXRel`
`DeviceToLogicalY`
`DeviceToLogicalYRel`
	Переводит физические координаты в логические, как в абсолютные (для позицирования), так и в относительные (для ширины и высоты).

`LogicalToDeviceX`
`LogicalToDeviceXRel`
`LogicalToDeviceY`
`LogicalToDeviceYRel`
	Переводит логические координаты в физические, как в абсолютные (для позицирования), так и в относительные (для ширины и высоты).

`SetMapMode` `GetMapMode`	Как было рассказано ранее, данный методы определяют (вместе с `SetUserScale`) способ перевода логических единиц в физические.
`SetAxisOrientation`	Задает ориентацию для осей `x` и `y`: направление от наименьшего к наибольшему значению на оси. По умолчанию ось `x` сорентирована слева направо (`true`), а ось `y` — сверху вниз (`false`).

Таблица 5.6: Методы контекста устройства (продолжение)

Имя	Описание
SetDeviceOrigin GetDeviceOrigin	Устанавливает начало координат для контекста. Данный метод можно, например, использовать, чтобы поместить графику в заданное место на странице.
SetUserScale GetUserScale	Установить и получить масштаб, применяемый для перевода логических единиц в физические.

5.4 Написание текста

Указанный текст рисуется на контексте устройства методом DrawText текущим шрифтом, используя заданный фоновый режим (прозрачный или перекрывающий), а также текущий цвет для фона и текста. Если фоновым режимом является wxSOLID, то область под текстом закрашивается текущим фоновым цветом, а если wxTRANSPARENT, то текст рисуется без изменения фона.

В метод DrawText передается выводимая строка, а также два целых числа или объект класса wxPoint. Текст выводится так, чтобы его самая левая верхняя точка была на заданной позиции. Вот простой пример вывода строки:

```
// Рисуем текстовую строку на экране в заданной точке
void DrawTextString(wxDC& dc, const wxString& text,
                    const wxPoint& pt)
{
    wxFont font(12, wxFONTFAMILY_SWISS, wxNORMAL, wxBOLD);
    dc.SetFont(font);
    dc.SetBackgroundMode(wxTRANSPARENT);
    dc.SetTextForeground(*wxBLACK);
    dc.SetTextBackground(*wxWHITE);
    dc.DrawText(text, pt);
}
```

Вы также можете использовать метод контекста устройства DrawRotatedText для написания текста под определенным углом, задаваемым последним аргументом метода. Следующий код рисует в цикле текст, изменяя угол на 45 градусов. Результат работы программы вы можете увидеть на рисунке 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);
```

Рис. 5.2: Рисование текста под углом

В системе Windows можно поворачивать только TrueType-шрифты. Обратите внимание, что `wxNORMAL_FONT` не относится к такому типу шрифтов.

Часто разработчику необходимо узнать сколько места займет текст на контексте устройства. Для этого используется метод `GetTextExtent`, которому передаются два указателя на целочисленные переменные, в которые запишутся требуемые размеры. Прототип метода имеет вид:

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

Значения по умолчанию для последних аргументов означают, что вам необходимо получить только суммарную высоту или ширину занимаемые строкой. Если вы передадите дополнительные параметры, то сможете получить больше информации о размерности текста относительно базовой линии (это воображаемая линия на которой располагается текст).

Вот код, который центрирует строку в окне, используя `GetTextExtent`:

```
void CenterText(const wxString& text, wxDC& dc, wxWindow* win)
{
    // Устанавливаем шрифт, цвет фона и текста
    dc.SetFont(*wxNORMAL_FONT);
    dc.SetBackgroundMode(wxTRANSPARENT);
    dc.SetTextForeground(*wxRED);

    // Получаем размер окна и текста
    wxSize sz = win->GetClientSize();
    wxCoord w, h;
    dc.GetTextExtent(text, & w, & h);

    // Центрируем текст в окне, но никогда не пишем
    // текст по отрицательным координатам
```

```
    int x = wxMax(0, (sz.x - w)/2);
    int y = wxMax(0, (sz.y - h)/2);

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

Также можно вызвать метод `GetPartialTextExtents`, который получает ширину каждого символа строки и помещает результат в массив типа `wxArrayInt`, передаваемый по ссылке. Если вам необходима точная информация о ширине каждого отдельного символа строки, то данный способ предпочтителен, так как некоторые платформы выполняют его быстрее, чем с использованием вызова `GetTextExtent` для каждого символа.

5.5 Рисование линий и фигур

Простые примитивы для рисования включают в себя точки, линии, прямоугольники, окружности и элипсы. Текущая обводка определяет цвет линии или границы, а кисть определяет цвет внутренности. Например:

```
void DrawSimpleShapes(wxDC& dc)
{
    // Устанавливаем цвет линий в черный, а закрашиваем зеленым
    dc.SetPen(wxPen(*wxBLACK, 2, wxSOLID));
    dc.SetBrush(wxBrush(*wxGREEN, wxSOLID));

    // Рисуем точку
    dc.DrawPoint(5, 5);

    // Рисуем линию
    dc.DrawLine(10, 10, 100, 100);

    // Рисуем прямоугольник на (50, 50) с размерами (150, 100)
    // крестовой кистью
    dc.SetBrush(wxBrush(*wxBLACK, wxCROSS_HATCH));
    dc.DrawRectangle(50, 50, 150, 100);

    // Выбираем красную кисть
    dc.SetBrush(*wxRED_BRUSH);

    // Рисуем закругленный прямоугольник на (150, 20) с
    // размерами (100, 50) и радиусом закругления углов 10
    dc.DrawRoundedRectangle(150, 20, 100, 50, 10);

    // Рисуем другой закругленный прямоугольник без границы
    dc.SetPen(*wxTRANSPARENT_PEN);
    dc.SetBrush(wxBrush(*wxBLUE));
    dc.DrawRoundedRectangle(250, 80, 100, 50, 10);
```

```
    // Делаем обводку и кисть черными
    dc.SetPen(wxPen(*wxBLACK, 2, wxSOLID));
    dc.SetBrush(*wxBLACK);

    // Рисуем окружность на (100, 150) радиуса 60
    dc.DrawCircle(100, 150, 60);

    // Выбираем белую кисть
    dc.SetBrush(*wxWHITE);

    // Рисуем элипс, который закрашивает заданный прямоугольник
    dc.DrawEllipse(wxRect(120, 120, 150, 50));
}
```

В результате работы программы получается изображение на рисунке 5.3.

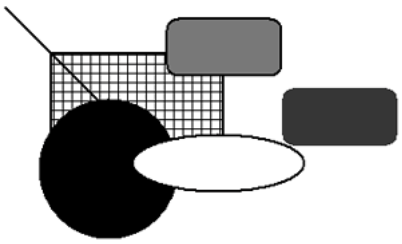

Рис. 5.3: Рисование простейших фигур

Обратите внимание, что последняя точка линии не нарисовалась.

Чтобы нарисовать круглую дугу используется метод `DrawArc`, принимающий в качестве параметров начальную, конечную и центральную точки. Дуга рисуется против часовой стрелки из начальной к конечной точке. Например,

```
int x = 10, y = 200, radius = 20;
dc.DrawArc(xradius, y, x + radius, y, x, y);
```

Данный код выводит дугу, изображенную на рисунке 5.4.

Рис. 5.4: Круглая дуга

Для рисования элиптической дуги используется метод `DrawEllipticArc`, который принимает на вход координаты и размеры прямоугольника, который содержит необходимую дугу, а также координаты начальной и конечной точки в градусах, отсчитанных от трехчасовой стрелки от центра прямоугольника. Если начальная и конечная точка совпадают, то рисуется эллипс. Следующий код рисует дугу, изображенную на рисунке 5.5.

```
// Рисует эллиптическую дугу внутри прямоугольника
// с координатами (10, 100) и размерами 200x40. Дуга
// расположена между 270 и 420 градусами.
dc.DrawEllipticArc(10, 100, 200, 40, 270, 420);
```

Рис. 5.5: Эллиптическая дуга

Если вам необходимо быстро нарисовать множество линий, то использование `DrawLines` будет намного эффективней вызова `DrawLine` множества раз. Следующий код рисует линии между десятью точками со смещением (100, 100).

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

int offsetX = 100;
int offsetY = 100;

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

`DrawLines` не закрашивает область, ограниченную линиями. Для рисования закрашенных фигур с произвольным числом сторон используют `DrawPolygon`, а для некоторых — `DrawPolyPolygon`. `DrawPolygon` получает количество точек и массив, смещение фигуры относительно начала координат, а также стиль заполнения: `wxODDEVEN_RULE` (по умолчанию) или `wxWINDING_RULE`. `DrawPolygonPolygon` дополнительно получает массив целых чисел, который определяет число точек в каждом полигоне.

Следующий фрагмент демонстрирует рисование полигона и мультиполигона, получая результат, показанный на рисунке 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);
}
```

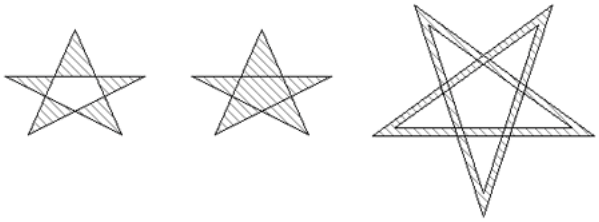

Рис. 5.6: Рисование полигонов

5.6 Рисование сплайнов

`DrawSpline` позволяет вам рисовать кривые, известные как «сплайны» между множеством точек. Есть версия для трех точек и для произвольного числа точек. Использование обеих версий проиллюстрировано в следующем примере:

```
// Рисуем сплайн через три точки
dc.DrawSpline(10, 100, 200, 200, 50, 230);

// Рисуем сплайн через пять точек
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);
```

Результатом работы данного кода являются два сплайна, изображенные на рисунке 5.7.

Рис. 5.7: Рисование сплайнов

5.7 Рисование битовых карт

Существует для основных метода рисования битовую карту в контексте устройства: `DrawBitmap` и `Blit`. `DrawBitmap` является упрощенной формой `Blit` и получает в качестве параметров битовую карту, положение и булевый флаг, определяющий рисование с поддержкой прозрачности. В зависимости от того как битовая карта была создана или загружена, прозрачность может задаваться как в виде простой маски, так и с помощью альфа-канала (который реализует полупрозрачность). Следующий код загружает изображение с альфа-каналом и рисует его поверх строк с текстом:

```
wxString msg = wxT("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);
```

Код рисует на экран изображение, показанное на рисунке 5.8, где тень на битовой карте выглядит как бы падающей на текст.

Рис. 5.8: Рисование с прозрачностью

Метод `Blit` гораздо более гибкая и позволяет вам скопировать выделенную часть из одного контекста устройства в другой. Вот ее прототип:

```
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);
```

Данный метод позволяет скопировать некоторую область из контекста `dcSource` в текущий контекст (именно с ним и работает данный метод). Область, определяемая шириной (`width`) и высотой (`height`) рисуется, начиная с заданных координат `(destX, destY)`. Начальная позиция в источнике задается с помощью `(srcX, srcY)`. Логическая функция `logicalFunc` обычно равна `wxCOPY`, что означает, что биты пересылаются без преобразования. Заметим, что некоторые платформы могут не поддерживать никаких логических функций, кроме `wxCOPY`. За дополнительной информацией обратитесь к разделу «Логические функции» данной главы.

Последние три параметра используются только в случае, когда контекст-источник имеет тип `wxMemoryDC` и содержит битовую карту с прозрачностью. `useMask` определяет нужно ли делать вывод с учетом прозрачности, а `srcMaskX` и `srcMaskY` активируют вывод с учетом прозрачности начиная с некоторой заданной точки.

Следующий код загружает небольшую палитру в битовую карту и использует `Blit` для заполнения большой битовой карты, используя прозрачность там, где это возможно.

```
wxMemoryDC dcDest;
wxMemoryDC dcSource;

int destWidth = 200, destHeight = 200;

// Создаем битовую карту для результата
wxBitmap bitmapDest(destWidth, destHeight);

// Загружаем битовую карту с палитрой
wxBitmap bitmapSource(wxT("pattern.png"), wxBITMAP_TYPE_PNG);

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

// Делаем фон белым
dcDest.SelectObject(bitmapDest);
dcDest.SetBackground(*wxWHITE_BRUSH);
dcDest.Clear();

dcSource.SelectObject(bitmapSource);

// Замостить большую битовую карту содержимым маленькой
```

```
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);
    }

// Освобождаем ресурсы
dcDest.SelectBitmap(wxNullBitmap);
dcSource.SelectBitmap(wxNullBitmap);
```

Используя метод `DrawIcon` можно рисовать иконки напрямую. Эта операция всегда предполагает использование прозрачности. Например,

```
#include "file.xpm"

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

5.8 Заливка произвольных областей

`FloodFill` используется для заливки произвольных областей контекста устройства, ограниченных некоторой цветовой границей. Начиная с заданной точки алгоритм использует указанный вами цвет в качестве границы заполнения. Контекст устройства заливается текущей кистью.

В следующем примере рисуется зеленый прямоугольник с красной границей, после чего он последовательно заливается черным и синим цветом.

```
// Рисуем зеленый прямоугольник с красной границей
dc.SetPen(*wxRED_PEN);
dc.SetBrush(*wxGREEN_BRUSH);

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

// Заливаем зеленый цвет черным (пока можем найти зеленый цвет)
dc.FloodFill(50, 50, *wxGREEN, wxFLOOD_SURFACE);
dc.SetBrush(*wxBLUE_BRUSH);

// Далее заливаем синим (до красной границы)
dc.FloodFill(50, 50, *wxRED, wxFLOOD_BORDER);
```

Вызов данного метода может завершиться с ошибкой, если он не может найти указанного цвета или если указанная точка находится за областью отсечения. Метод `FloodFill` не работает с контекстом принтера и с `wxMetafileDC`.

5.9 Логические функции

Логическая функция определяет как пиксель с источника (цвета обводки или кисти, а также из контекста устройства в случае использования `Blit`) будет комбинироваться с пикселем в точке назначения в текущем контексте устройства. По умолчанию используется логическая функция `wxCOPY`, которая означает, что пиксель просто зарисовывается текущим цветом. Все остальные функции комбинируют текущий цвет и цвет фона, используя некоторую логическую операцию. `wxINVERT` часто используется в реализациях различных перемещений фигур и линий, так как при ее использовании рисование фигуры на том же месте еще раз полностью стирает изображение данной фигуры.

В следующем примере рисуется пунктирная линия, а далее она стирается с помощью `wxINVERT` для восстановления старого изображения, после чего восстанавливается обычный режим рисования.

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

// Рисуем линию
dc.SetLogicalFunction(wxINVERT);
dc.DrawLine(10, 10, 100, 100);

// Рисуем еще раз, уничтожая старое изображение
dc.DrawLine(10, 10, 100, 100);

// Восстанавливаем обычный режим рисования
dc.SetLogicalFunction(wxCOPY);
```

Еще одним использованием логических функций является получение нового изображения, посредством комбинирования существующих.

В таблице 5.7 возможные значения логических функций, а также их смысл.

Таблица 5.7: Логические функции

Логическая функция	Значение (src = источник, dst = назначение)
`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)

Таблица 5.7: (продолжение)

Логическая функция	Значение (src = источник, dst = назначение)
wxSET	1
wxSRC_INVERT	NOT src
wxXOR	src XOR dst

5.10 Использование инструментария для печати

Как уже упоминалось ранее, для печати можно напрямую использовать контекст wxPrinterDC. Однако существует более гибкий метод печати, который позволяет вам «управлять» процессом печати. Для этого разработчику необходимо создать наследника от wxPrintout, перегрузив методы, определяющие печать страниц (OnPrintPage), количество печатаемых страниц (GetPageInfo), настройки документа (OnPreparePrinting) и так далее. wxWidgets сама покажет диалог печати, создаст необходимый контекст и позовет соответствующие методы wxPrintout, когда это потребуется. Созданный класс может быть одновременно использован и для печати, и для предпросмотра.

Для старта печати необходимо передать объект wxPrintout в объект класса wxPrinter и вызвать метод Print, чтобы запустить процесс печати, а также при необходимости показать диалог печати. Например,

```
// Глобальный объект для хранения настроек печати
wxPrintDialogData g_printDialogData;

// Обработчик для пункта меню "Печать"
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.\n\
Perhaps your current printer is not set correctly?"),
                         wxT("Printing"), wxOK);
        else
            wxMessageBox(wxT("You cancelled printing"),
                         wxT("Printing"), wxOK);
    }
    else
    {
        (*g_printDialogData) = printer.GetPrintDialogData();
    }
}
```

Так как метод `Print` завершается только когда все страницы сформированы и посланы на принтер, то объект для печати можно создавать на стеке.

Класс `wxPrintDialogData` хранит данные, которые пользователь выбрал в диалоге печати, такие как номер страницы для печати и число необходимых копий. Достаточно неплохим решением является создание глобального объекта `wxPrintDialogData`, который будет хранить настройки, установленные пользователем в последний раз. Вы можете передать указатель на объект `wxPrintDialogData` в `wxPrinter`, чтобы они учитывались в показываемом диалоге, а после печати опять копировать их из `wxPrinter` в ваш глобальный объект, как это сделано в прошлом примере (в реальном приложении `g_printDialogData` возможно стоит сделать членом вашего класса приложения). Обратитесь к Главе 8 «Использование стандартных диалогов», чтобы узнать больше о диалогах печати, настройке страниц и их использовании.

Для предпросмотра документа создайте объект `wxPrintPreview`, передав в него два объекта печати. Первый используется непосредственно для предпросмотра, а второй когда пользователь захочет что-то распечатать. Также существует возможность передать объект класса `wxPrintDialogData`, чтобы предпросмотр взял настройки печати, которые пользователь выбрал ранее. После передачи объектов в `wxPreviewFrame` необходимо вызвать метод `Initialize`, который покажет окно предпросмотра. Пример:

```
// Обработчик для пункта меню "Предпросмотр"
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.\n\
Perhaps 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);
}
```

После инициализации фрейм предпросмотра блокирует все остальные окна, чтобы блокировать возможность действий, которые могут изменить документ после начала процесса печати или предпросмотра. Закрытие данного фрейма автоматически уничтожит и два связанных с ним объекта печати. На рисунке 5.9

показано окно предпросмотра с кнопками, позволяющими осуществлять навигацию по документу, печатать, а также приближать и отдалять документ.

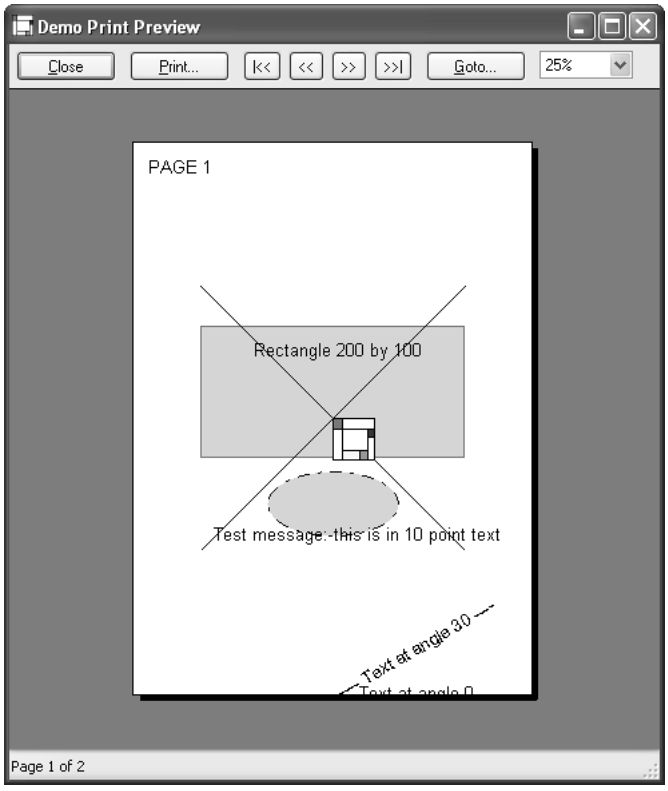

Рис. 5.9: Окно предпросмотра

5.11 Углубленное знакомство с wxPrintout

При создании объекта класса `wxPrintout` приложение может дополнительно задать заголовок, который будет показываться в менеджере печати под некоторыми операционными системами. Вам также необходимо перегрузить по крайней мере `GetPageInfo`, `HasPage` и `OnPrintPage`, и, возможно, некоторые другие методы, перечисленные ниже.

`GetPageInfo` должен быть перезагружен, чтобы возвратить правильные значения для `minPage`, `maxPage`, `pageFrom` и `pageTo`. Первые два целочисленные значения определяют интервал страниц, поддерживаемый объектом печати. Вторые два значения содержат выбор пользователя (в данный момент не используется в wxWidgets). Значения по умолчанию для `minPage` и `maxPage` — `1` и `32000`, соответственно. Однако, печать будет остановлена в любом случае, если `HasPage` возвратит `false`. Обычно ваш метод `OnPreparePrinting` будет отвечать за вычисление значений, возвращаемых `GetPageInfo` и выглядеть как-то так:

```
void MyPrintout::GetPageInfo(int *minPage, int *maxPage,
                             int *pageFrom, int *pageTo)
{
```

```
    *minPage = 1; *maxPage = m_numPages;
    *pageFrom = 1; *pageTo = m_numPages;
}
```

HasPage должен возвращать false, если аргумент находится за пределам допустимого номера страницы. Очень часто реализация данного метода выглядит следующим образом (где m_numPages вычисляется в методе OnPreparePrinting):

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

OnPreparePrinting вызывается перед началом процесса печати или предпросмотра и обычно предназначен для выполнения необходимых подготовительных шагов, включая вычисление числа страниц в документе. OnPreparePrinting для получения необходимой информации может вызывать методы класса wxPrintout, такие как GetDC, GetPageSizeMM, IsPreview и так далее.

OnBeginDocument вызывается в начале печати каждого документа. Этот метод получает в качестве параметра номера начальной и конечной страниц. Если этот метод переопределяется, то он должен вызвать базовую функцию wxPrintout::OnBeginDocument, а также (если необходимо) wxPrintout::OnEndDocument.

OnBeginPrinting вызывается единожды в начале каждого цикла печати (относительно числа копий), а OnEndPrinting вызывается в конце.

В OnPrintPage передается номер страницы и приложение должно переопределить данный метод и возвращать true, если печать страницы закончилась удачно. Возвращение false прервет процесс печати. Данный метод обычно использует wxPrintout::GetDC, чтобы получить контекст, на котором осуществляется рисование.

Далее приводится список дополнительных методов, которые как правило не перегружаются и которые необходимо использовать в своих перегруженных методах.

IsPreview используется для определения того, идет ли сейчас печать или предпросмотр.

GetDC возвращает подходящий контекст устройства для текущей задачи. При печати возвращается wxPrinterDC, а при предпросмотре возвращается wxMemoryDC, так как предпросмотр использует битовую карту в контексте устройства в памяти.

GetPageSizeMM возвращает размер страницы принтера в миллиметрах, тогда как GetPageSizePixels возвращает размер в пикселях (максимальное разрешение принтера). При предпросмотре последняя величина не совпадает с размером, возвращаемым wxDC::GetSize, который относится к битовой карте для предпросмотра.

GetPPIPrinter возвращает число пикселей на логический дюйм для текущего контекста устройства, а GetPPIScreen возвращает число пикселей на логический дюйм для экрана.

5.12 Масштабирование при печати и предпросмотре

При рисовании в окне вас, возможно, не беспокоит масштаб вашей графики, так как дисплей имеет похожее разрешение. Однако есть несколько нюансов на которые вам необходимо обращать внимание при выводе изображения на принтер:

Вы должны масштабировать и правильно размещать изображение на странице, чтобы оно влезло. В случае необходимости графику придется разбить на несколько страниц.

Шрифты основаны на разрешении экрана, а поэтому при печати текста необходимо правильно выбрать масштаб, чтобы разрешение принтера совпало с разрешением на мониторе. Деление разрешения принтера (`GetPPIPrinter`) на разрешение монитора (`GetPPIScreen`) обычно дает подходящий масштаб для вывода текста.

Для создания предпросмотра wxWidgets использует `wxMemoryDC`, который рисует в памяти. Размер битовой карты (возвращаемой `wxDC::GetSize`) зависит от текущего увеличения изображения, поэтому необходимо вычислять дополнительный масштабирующий коэффициент, чтобы правильно обработать такую ситуацию. Разделите размер, возвращаемый `GetSize` на размер настоящей страницы, возвращаемый `GetPageSizePixels` для получения масштабирующего коэффициента. Эта величина должна умножаться на любой другой коэффициент, который вы будете использовать.

Использование `wxDC::SetUserScale` позволяет контексту устройства выполнить масштабирование для всех дальнейших графических операций, а `wxDC::SetDeviceOrigin` устанавливает начало координат (например, в центре графики на экране). Допускается многократный вызов функции изменения масштаба и установки начала координат при рисовании на одной странице.

В дистрибутиве библиотеки wxWidgets существует пример samples/printing, который показывает как правильно делать масштабирование. Далее приводится немного модифицированный код из данного примера, который масштабирует и помещает графику размером 200 на 200 пикселей в диалог предпросмотра и на принтер.

```
void MyPrintout::DrawPageOne(wxDC *dc)
{
    // Вы можете использовать этот код, если вам необходимо
    // промасштабировать графику с известным размером на страницу

    // В данном примере мы знаем, что графика имеет размер 200x200.
    // Однако в реальном приложении необходимо произвести вычисления,
    // чтобы получить корректные размеры.
    float maxX = 200;
    float maxY = 200;

    // Предположим нам хочется сделать смещение для графики
    // в 50 логических единиц
    float marginX = 50;
    float marginY = 50;
```

```
    // Добавить смещение к размеру графики
    maxX += (2*marginX);
    maxY += (2*marginY);

    // Получаем размер контекста в пикселях
    int w, h;
    dc->GetSize(&w, &h);

    // Вычисляем приемлемый коэффициент масштабирования
    float scaleX=(float)(w/maxX);
    float scaleY=(float)(h/maxY);

    // Вычисляем максимально большой масштаб
    float actualScale = wxMin(scaleX,scaleY);

    // Вычисляем позицию на контексте, чтобы центрировать графику
    float posX = (float)((w - (200*actualScale))/2.0);
    float posY = (float)((h - (200*actualScale))/2.0);

    // Устанавливаем масштаб и смещение
    dc->SetUserScale(actualScale, actualScale);
    dc->SetDeviceOrigin( (long)posX, (long)posY );

    // Рисуем
    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);
}
```

В коде мы использовали метод `wxDC::GetSize`, чтобы получить разрешение для

предпросмотра или принтера, благодаря чему смогли разместить изображение на странице. Нас не интересовало разрешение в точках на дюйм принтера, как в случае, если бы нам захотелось написать текст или нарисовать строку заданного в миллиметрах размера. Поэтому наша графика не имеет точного размера — мы просто стремились к тому, чтобы изображение вместилось на страницу.

Следующий пример показывает как написать строку и нарисовать линию, чтобы их размер равнялся в точности их размерам на экране монитора.

```
void MyPrintout::DrawPageTwo(wxDC *dc)
{
    // Вы можете использовать данный код, чтобы текст на
    // принтере в точности соответствовал своему изображению на экране.
    // Этот код также рисует линию длинной 5 сантиметров.

    // Берем логические единицы в пикселях на дюйм для экрана и принтера
    int ppiScreenX, ppiScreenY;
    GetPPIScreen(&ppiScreenX, &ppiScreenY);
    int ppiPrinterX, ppiPrinterY;
    GetPPIPrinter(&ppiPrinterX, &ppiPrinterY);

    // Получаем пропорцию, которая служит грубым приближением
    // пропорции между разрешением экрана и принтера
    float scale = (float)((float)ppiPrinterX/(float)ppiScreenX);

    // Теперь проверяем случай, когда реальный размер страницы
    // был уменьшен (например, при рисовании при предпросмотре
    // на контексте в памяти)
    int pageWidth, pageHeight;
    int w, h;
    dc->GetSize(&w, &h);
    GetPageSizePixels(&pageWidth, &pageHeight);

    // Если ширина страницы принтера равна ширине текущего контекста,
    // то ничего не меняем. Но w должно быть шириной битовой карты для
    // предпросмотра, поэтому уменьшаем масштаб.
    float overallScale = scale * (float)(w/(float)pageWidth);
    dc->SetUserScale(overallScale, overallScale);

    // Вычисляет множитель для преобразования миллиметров в логические
    // единицы. В одном дюйме приблизительно 25.4 мм. У нас ppi единиц
    // устройства на дюйм. Поэтому 1 мм соответствует ppi/25.4 единицам
    // устройства. Мы также делим на число на множитель преобразования
    // экран-на-принтер, так как нам необходимо сделать так, чтобы иметь
    // возможность передать логические единицы в DrawLine.

    // Рисуем фигуру размером 50 мм на 50 мм
    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 );
}
```

5.12.1 Печать в Unix с использованием GTK+

В отличие от Mac OS X и Windows, в системе Unix нет единого стандартного механизма для отображения текста и графики на экране и его печати. Вместо этого вывод на экран осуществляется с помощью библиотеки X11 (через GTK+ и wxWidgets), а печать через посылку файла с PostScript-командами на принтер. Работа со шрифтами в последнем случае очень сложна, из-за чего существует не очень много WYSIWYG (сокращение от What You See Is What You Get[1]) редакторов под эту операционную систему. В прошлом wxWidgets предоставлял только свою собственную реализацию печати, использующую PostScript, которая не давала гарантии полного совпадения изображения на мониторе и на экране.

С версии 2.8 проект GNOME Free Software Desktop реализует поддержку печати через свои библиотеки `libgnomeprint` и `libgnomeprintui` для которых многие из указанных проблем решены. Начиная с версии 2.5.4 порт wxWidgets для GTK+ может использовать указанные библиотеки, если wxWidgets соответствующим образом сконфигурирована и если указанные библиотеки присутствуют в системе. Если при сборке wxWidgets указан ключ --with-gnomeprint, то при запуске приложение будет искать указанные библиотеки GNOME. Если они найдутся, то печать будет осуществляться через них, иначе приложение будет использовать старый код печати через PostScript. Заметим, что для запуска приложения не требуется, чтобы библиотеки печати GNOME были установлены, то есть не существует зависимости от указанных библиотек.

5.13 Трехмерная графика в wxGLCanvas

Будет нелишним упомянуть, что wxWidgets обладает способностью рисовать трехмерную графику, благодаря библиотеке OpenGL и классу `wxGLCanvas`. Данный класс можно также использовать с библиотекой Mesa (клон OpenGL), если ваша платформа по каким-то причинам не поддерживает OpenGL.

Для включения поддержки `wxGLCanvas` в системе Windows необходимо отредактировать файл include/wx/msw/setup.h, установив `wxUSE_GLCANVAS` в `1` и скомпилировать библиотеку с флагом `USE_OPENGL=1`. Возможно также потребуется

[1] что видишь, то и получишь (англ.)

добавить opengl32.lib в список библиотек, с которыми линкуется ваша программа. В системах Unix и Mac OS X необходимо передать --with-opengl в скрипт конфигурирования для компиляции с использованием OpenGL или Mesa.

Если вы уже программировали с использованием OpenGL, то перейти на использование `wxGLCanvas` будет для вас простой задачей. Вы создаете объект класса `wxGLCanvas` внутри вашего фрейма или любого другого окна, вызываете `wxGLCanvas::SetCurrent`, чтобы напрямую передавать следующие команды этому окну, вызываете обычные OpenGL-команды, а далее вызываете `wxGLCanvas::SwapBuffers`, чтобы показать содержимое OpenGL-буфера в окне.

Следующий обработчик сообщений иллюстрирует основные принципы рисования трехмерных изображений и рисует на экране куб. Полный код данного приложения вы можете найти в каталоге samples/opengl/cube вашего дистрибутива wxWidgets.

```
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);

    /* очищаем буфер цвета и глубины */
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    /* рисуем шесть граней куба */
    glBegin(GL_QUADS);
    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();
}
```

Рисунок 5.10 содержит результат другого примера OpenGL — пингвина, которого можно вращать, используя мышку. Данный пример расположен в каталоге samples/opengl/penguin.

Рис. 5.10: Пример использования OpenGL — «пингвин»

5.14 Итоги

В данной главе вы изучили рисование на контексте устройства, использование инструментария для печати, а также получили короткое введение в `wxGLCanvas`. В качестве примера рекомендуем изучить исходный код в дистрибутиве wxWidgets:

- samples/drawing
- samples/font
- samples/erase
- samples/image
- samples/scroll
- samples/printing
- src/html/htmprint.cpp

- demos/bombs
- demos/fractal
- demos/life

Если потребуются более продвинутые инструменты для рисования, то возможно стоит посмотреть библиотеку wxArt2D, которая в том числе умеет загружать и сохранять графические объекты, используя SVG-файлы (Scalable Vector Graphics), осуществлять обновления без мерцания, рисовать градиенты, векторные пути и так далее. Обратитесь к Дополнению E «Сторонние утилиты для wxWidgets» за информацией о получении wxArt2D.

В следующей главе мы расскажем как ваше приложение может взаимодействовать с мышью, клавиатурой и джойстиком.