6 Работа с базовыми составными блоками

Эта глава включает

- Использование контекста устройства для рисования на экране
- Добавление на фрейм оконных украшений
- Работа со стандартными диалогами выбора файлов и цвета
- Размещение виджетов и создание координатора
- Создание диалогов About и splash-окон

Даже простая программа на wxPython нуждается в применении стандартных элементов, например, таких как меню и диалогов. Все эти элементы выступают в роли составных блоков любого GUI-приложения, и вместе с такими виджетами как splash-окно (заставка), строка статуса (status bar) или диалог About, они обеспечивают более дружественную среду и придают Вашему приложению профессиональный вид и восприятие. В завершение первой части книги, мы обсудим с Вами создание программы, использующей все эти компоненты. Мы построим простую программу рисования, затем добавим эти блочные составные элементы и разъясним некоторые вопросы их использования. Мы закрепим фундаментальные понятия, затронутые в предшествующих главах, и в конечном счете у Вас получится простое, но в тоже время профессиональное приложение. Эта глава является промежуточным уровнем между базовыми понятиями, обсужденными в предшествующих главах и более подробным рассмотрением функциональности wxPython в частях 2 и 3.

Приложение, которое мы создадим в этой главе основано на примерах Doodle и Super Doodle, поставляемых вместе с wxPython в директории wxPython/samples. Это очень простая программа рисования, которая отслеживает указатель мыши и рисует линию при нажатии левой кнопки мыши. Рисунок 6.1 показывает простое исходное окно этой программы.

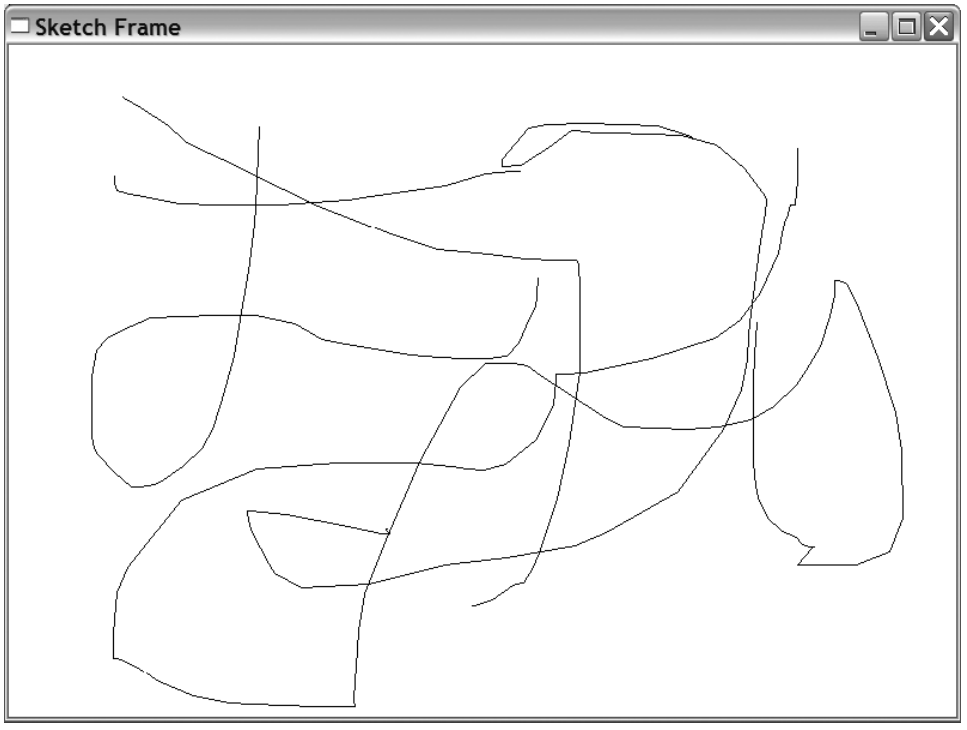

Рисунок 6.1 Образец окна без украшений

Мы выбрали такой пример, поскольку эта довольно простая программа иллюстрирует многие вопросы, возникающие при создании более сложных приложений. В этой главе мы покажем Вам, как нарисовать на экране линии, добавить панель состояния (status bar), панель инструментов (toolbar) и меню (menubar). Вы увидите, как можно использовать стандартные диалоги, такие как диалоги выбора файлов и цвета. Для размещения составных наборов виджетов мы используем координатор (sizer), а также добавим панель About и splash-окно. В конце главы Вы получите великолепную на вид заготовку программы.

6.1 Рисование на экране

Первоочередная задача Вашей программы для зарисовок состоит в рисовании на экране линии. Подобно другим GUI-инструментам, wxPython обеспечивает независимый от устройства набор инструментальных средств для рисования на различных типах дисплеев. В следующем разделе мы обсудим, как рисовать на экране.

6.1.1 Как рисовать на экране?

Для рисования на экране мы используем объект wxPython, называющийся контекстом устройства. Контекст устройства абстрагирует дисплейное устройство, предоставляя каждому устройству общий набор методов рисования, поэтому Ваш код рисования не зависит от типа целевого устройства. Контекст устройства в wxPython представлен абстрактным классом `wx.DC` и его подклассами. Так как `wx.DC` абстрактный класс, в своём приложении Вам нужно использовать один из его подклассов.

Использование контекста устройства

Таблица 6.1 отображает справочник по подклассам `wx.DC` и их использованию. Контексты устройств, которые используются для рисования в виджетах wxPython, должны всегда создаваться локально в виде временных объектов и не должны храниться между вызовами метода в виде глобальной переменной или другим способом. На некоторых платформах контексты устройства - это ограниченный ресурс и такое удержание ссылки на `wx.DC` может сделать программу нестабильной. Из-за способа, которым wxPython непосредственно использует контексты устройства, подклассы `wx.DC` имеют несколько тонких различий, которые учитываются при рисовании в виджетах. В главе 12 эти различия объясняются более подробно.

Таблица 6.1 Краткое руководство по подклассам контекста устройства `wx.DC`

Контекст устройства	Использование
`wx.BufferedDC`	Используется для буферизации ряда команд рисования, до их завершения и готовности к дальнейшему рисованию на экране. Это предотвращает нежелательное мерцание при отображении.
`wx.BufferedPaintDC`	Тоже, что и `wx.BufferedDC`, но используется только в пределах `wx.PaintEvent`. Создавайте только временные экземпляры этого класса.
`wx.ClientDC`	Используется для рисования в оконном объекте. Применяйте этот класс, когда Вам необходимо рисовать в основной области виджета, т.е. не на границе или любых элементах оформления. Основная область иногда называется клиентской областью, отсюда и имя этого DC. Класс `wx.ClientDC` должен создаваться только временно. Этот класс используется только вне тела `wx.PaintEvent`.
`wx.MemoryDC`	Используется для рисования графики на сохраненном в памяти битовом изображении (bitmap), без его отображения. Вы можете затем выбрать битовое изображение и использовать метод `wx.DC.Blit()`, чтобы нарисовать его в окне.
`wx.MetafileDC`	В операционных системах Windows этот контекст устройства позволяет Вам создавать стандартные метафайлы данных.
`wx.PaintDC`	Идентичен контексту `wx.ClientDC`, за исключением того, что он используется только в пределах `wx.PaintEvent`. Создавайте только временные экземпляры этого класса.
`wx.PostScriptDC`	Используется для записи инкапсулированных файлов PostScript.
`wx.PrinterDC`	Используется в операционных системах Windows для вывода на принтер.
`wx.ScreenDC`	Используется для рисования непосредственно на экране, поверх и вне любого отображаемого окна. Этот класс должен создаваться только временно.
`wx.WindowDC`	Используется для рисования на всей области оконного объекта, включая границу и любые другие элементы оформления, не относящиеся к клиентской области. Операционные системы отличные от Windows могут не поддерживать этот класс.

Листинг 6.1 содержит начальный код изображенного на рисунке 6.1 окна рисования. Поскольку этот код показывает приёмы рисования в контексте устройства, мы его подробно прокомментируем.

Листинг 6.1 Начальный код окна рисования

```
import wx

class SketchWindow(wx.Window):
    def __init__(self, parent, ID):
        wx.Window.__init__(self, parent, ID)
        self.SetBackgroundColour("White")
        self.color = "Black"
        self.thickness = 1
        self.pen = wx.Pen(self.color, self.thickness, wx.SOLID)
        self.lines = []
        self.curLine = []
        self.pos = (0, 0)
        self.InitBuffer()
        self.Bind(wx.EVT_LEFT_DOWN, self.OnLeftDown)
        self.Bind(wx.EVT_LEFT_UP, self.OnLeftUp)
        self.Bind(wx.EVT_MOTION, self.OnMotion)
        self.Bind(wx.EVT_SIZE, self.OnSize)
        self.Bind(wx.EVT_IDLE, self.OnIdle)
        self.Bind(wx.EVT_PAINT, self.OnPaint)

    def InitBuffer(self):
        size = self.GetClientSize()
        self.buffer = wx.EmptyBitmap(size.width, size.height)
        dc = wx.BufferedDC(None, self.buffer)
        dc.SetBackground(wx.Brush(self.GetBackgroundColour()))
        dc.Clear()
        self.DrawLines(dc)
        self.reInitBuffer = False

    def GetLinesData(self):
        return self.lines[:]

    def SetLinesData(self, lines):
        self.lines = lines[:]
        self.InitBuffer()
        self.Refresh()

    def OnLeftDown(self, event):
        self.curLine = []
        self.pos = event.GetPositionTuple()
        self.CaptureMouse()

    def OnLeftUp(self, event):
        if self.HasCapture():
            self.lines.append((self.color,
                self.thickness,
                self.curLine))
            self.curLine = []
```

Создание объекта wx.Pen

1

Присоединение событий

2

Создание контекста устройства с буферизацией

3

4

Использование контекста устройства

5 **Получение позиции мыши**

6 **Захват мыши**

```
            self.ReleaseMouse()
    def OnMotion(self, event):
        if event.Dragging() and event.LeftIsDown():
            dc = wx.BufferedDC(wx.ClientDC(self), self.buffer)
            self.drawMotion(dc, event)
        event.Skip()

    def drawMotion(self, dc, event):
        dc.SetPen(self.pen)
        newPos = event.GetPositionTuple()
        coords = self.pos + newPos
        self.curLine.append(coords)
        dc.DrawLine(*coords)
        self.pos = newPos

    def OnSize(self, event):
        self.reInitBuffer = True

    def OnIdle(self, event):
        if self.reInitBuffer:
            self.InitBuffer()
            self.Refresh(False)

    def OnPaint(self, event):
        dc = wx.BufferedPaintDC(self, self.buffer)

    def DrawLines(self, dc):
        for colour, thickness, line in self.lines:
            pen = wx.Pen(colour, thickness, wx.SOLID)
            dc.SetPen(pen)
            for coords in line:
                dc.DrawLine(*coords)

    def SetColor(self, color):
        self.color = color
        self.pen = wx.Pen(self.color, self.thickness, wx.SOLID)

    def SetThickness(self, num):
        self.thickness = num
        self.pen = wx.Pen(self.color, self.thickness, wx.SOLID)

class SketchFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, "Sketch Frame",
            size=(800,600))
        self.sketch = SketchWindow(self, -1)

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = SketchFrame(None)
    frame.Show(True)
    app.MainLoop()
```

7 Освобождение мыши

8 Определение продолжения операции перетаскивания

9 Создание другого контекста с буферизацией

10 Рисование в контексте устройства

11 Обработка события изменения размера

12 Обработка простоя

13 Обработка запроса на прорисовку

14 Рисование всех линий

1. Экземпляр `wx.Pen` определяет цвет, толщину и стиль нарисованных в контексте устройства линий. Стили отличные от `wx.SOLID` включают `wx.DOT`, `wx.LONGDASH` и `wx.SHORTDASH`.

2. Для рисования этому окну необходимо отвечать на различные типы событий мыши. Оно реагирует на нажатие и отпускание левой кнопки мыши, движение мыши, изменение размеров окна и перерисовку окна. Также определяется обработка, которая происходит в течение простоя.

3. Контекст устройства с буферизацией создается в два этапа: (1) Создается пустое битовое изображение, которое служит в качестве внеэкранного буфера и (2) На основе внеэкранного буфера создается буферизованный контекст устройства. Буферизованный контекст используется для того, чтобы избежать перерисовки проведенных линий рисунка и исключить вызываемое этим мерцание экрана. Позже в этом подразделе мы подробнее обсудим контексты устройства с буферизацией.

4. Эти строки инициируют команды рисования в контексте устройства, а именно – устанавливается кисть заполнения фона и очищается устройство. Объект `wx.Brush` определяет цвет и стиль фона для команд заливки.

5. Метод `GetPositionTuple()` объекта `event` возвращает кортеж Python, содержащий точную позицию, в которой выполнен щелчок мыши.

6. Метод `CaptureMouse()` направляет весь ввод мыши в данное окно, даже если Вы перемещаете мышь за пределами границы окна. Этот вызов должен отменяться последующим вызовом в программе метода `ReleaseMouse()`.

7. Вызов `ReleaseMouse()` возвращает систему в состояние предшествующее вызову `CaptureMouse()`. Для отслеживания окон, которые захватили мышь, приложение wxPython использует стек, а вызов `ReleaseMouse()` эквивалентен высвобождению этого стека. Это подразумевает, что Вам нужно равное количество вызовов `CaptureMouse()` и `ReleaseMouse()`.

8. Эта строка определяет, является ли событие перемещения мыши частью процесса проведения линии, характеризующегося происходящим событием движения мыши при ее нажатой левой кнопке. Методы `Dragging()` и `LeftIsDown()` являются методами `wx.MouseEvent` и возвращают True, если во время перемещения данное условие выполняется.

9. Поскольку `wx.BufferedDC` – один из временно создающихся контекстов устройства, перед тем как мы нарисуем линии, нам нужно создать другой контекст. В данном случае, в качестве основного контекста устройства мы создаем новый `wx.ClientDC` и снова используем в качестве буфера наш экземпляр переменной битового изображения.

10. Эти строки фактически используют контекст устройства для рисования на экране только что проведенной линии. Сначала, мы создаем кортеж координат `coords`, который является комбинацией кортежей `self.pos` и `newPos`. Здесь,

новая точка поступает из события `GetPositionTuple()`, а старая точка была определена при последнем вызове `OnMotion()`. Мы сохраняем этот кортеж в списке `self.curLine`, а затем используем функциональный вызов с синтаксисом распаковки (unpack syntax) для вызова `DrawLine()` с элементами кортежа в качестве аргументов. Метод `DrawLine()` принимает в качестве параметров (x1, y1, x2, y2) и рисует линию из точки (x1, y1) в точку (x2, y2). Частота, с которой происходит событие перемещения и появляется новая точка линии, зависит от базовой скорости системы.

11 Если размеры окна изменились, мы отмечаем это, сохраняя величину True в атрибуте `self.reInitBuffer`. Фактически же, мы ничего не делаем до следующего события простоя (idle event).

12 Когда придет событие простоя, приложение получает возможность реагировать на одно или несколько имевших место событий изменения размеров. Причина, по которой мы реагируем на событие простоя, а не на собственно событие изменения размеров, состоит в том, чтобы разрешить многократным событиям изменения размеров проследовать друг за другом максимально быстро без необходимости каждый раз выполнять перерисовку.

13 Обработка запроса на перерисовку изображения удивительно проста: для рисунка создается буферизованный контекст устройства. Создается реальный `wx.PaintDC` (так как мы находимся в обработчике прорисовки, нам нужен именно `wx.PaintDC`, а не экземпляр `wx.ClientDC`), и затем, после удаления экземпляра `dc`, битовое изображение блиттируется (копируется) в него. Более подробная информация о буферизации приведена в следующих разделах.

14 Этот метод используется, когда в результате изменения размеров (и последующей загрузки из файла) приложению необходимо перерисовать линии из экземпляра данных. Кроме того, мы используем надстройку `DrawFunc()`. В этом случае, мы проходим список линий, хранящийся в атрибуте coords, повторно создаем перо для каждой линии (скоро будет добавлена поддержка изменения характеристик пера), а затем рисуем все принадлежащие этой линии координатные кортежи.

Использование буфера рисования в этот пример обеспечивается двумя специальными подклассами `wx.DC`. Буфер рисования – это неотображаемая область, где все Ваши базовые команды рисования выполняются поочередно, а затем копируются на экран за один шаг. Преимущество буфера - в том, что пользователь не замечает отдельно происходящих команд рисования, и таким образом, экран обновляется с меньшим мерцанием. По этой причине, буферизация обычно используется при анимации или когда чертеж состоит из множества мелких частей.

В wxPython имеется два класса, которые используются для буферизации: `wx.BufferDC` обычно используется для буферизации `wx.ClientDC`, а `wx.BufferPaintDC` используется для буферизации `wx.PaintDC`. Работают они

практически одинаково. Буферизованный контекст устройства создается двумя аргументами. Первый аргумент – это контекст целевого устройства соответствующего типа (например, в строке (9) листинга 6.1, это новый экземпляр `wx.ClientDC`). Второй аргумент – это объект `wx.Bitmap`. В листинге 6.1, мы создаем битовое изображение при помощи функции `wx.EmptyBitmap`. Когда команды рисования относятся к буферизованному контексту устройства, для рисования битового изображения используется встроенный контекст `wx.MemoryDC`. При уничтожении буферного объекта, деструктор C++ использует метод `Blit()`, чтобы автоматически скопировать битовое изображение на целевое устройство. В wxPython уничтожение обычно происходит, когда объект покидает область действия. Поэтому буферизованные контексты устройства полезны только при их временном создании, при котором они уничтожаются, и выполняется блитирование (blit).

Например, в методе `OnPaint()` листинга 6.1, битовое изображение `self.buffer` уже определено. Буферный объект необходимо просто создать, устанавливая этим самым связь между существующим битовым изображением и временным контекстом `wx.PaintDC()` окна. Метод завершается, буферизованный DC немедленно покидает область, запускается деструктор и битовое изображение копируется на экран.

Функции контекста устройства

При использовании контекстов устройства не забывайте использовать корректный контекст в зависимости от вида выполняемого Вами рисования (особо помните различие между `wx.PaintDC` и `wx.ClientDC`). Только в случае, когда у Вас будет корректный контекст устройства, Вы сможете что-либо с ним делать. Таблица 6.2 перечисляет некоторые наиболее интересные методы `wx.DC`.

Таблица 6.2 Методы общего назначения `wx.DC`

Функция	Назначение
`Blit(xdest, ydest, width, height, source, xsrc, ysrc)`	Копирует биты непосредственно из контекста устройства `source` (источник) в контекст устройства, выполняющий данный вызов (приемник). Параметры `xdest` и `ydest` являются начальной точкой копии в контексте приемника. Следующие два параметра определяют ширину и высоту области копирования. Параметр `source` является исходным контекстом устройства, а `xsrc` и `ysrc` – начальная точка копии в этом контексте. Имеются дополнительные параметры для определения логической оверлейной функции и маски.

`Clear()`	Очищает контекст устройства, закрашивая всю его область текущей кистью фона.
`DrawArc(x1, y1, x2, y2, xc, yc)`	Рисует дугу окружности из начальной точки (x1, y1) в конечную точку (x2, y2). Точка (xc, yc) - центр окружности, которой принадлежит рисуемая дуга. Результирующая дуга заполняется текущей кистью. Эта функция рисует дугу из начальной точки в конечную точку против часовой стрелки. Имеется родственный метод `DrawEllipticalArc()`.
`DrawBitmap(bitmap, x, y, transparent)`	Копирует объект `wx.Bitmap`, начиная с точки (x, y). Если параметр `transparent` установлен в True, битовое изображение будет нарисовано прозрачным.
`DrawCircle(x, y, radius)` `DrawCircle(point, radius)`	Рисует окружность с центром в указанной точке и указанным радиусом. Имеется родственный метод `DrawEllipse`.
`DrawIcon(icon, x, y)`	Рисует объект `wx.Icon` в точке контекста (x, y).
`DrawLine(x1, y1, x2, y2)`	Рисует линию из точки (x1, y1) в точку (x2, y2). Имеется смежный метод `DrawLines()`, который берет список точек, состоящий из объектов Python `wx.Point` и соединяет их.
`DrawPolygon(points)`	Рисует полигон (многоугольник) на основании списка Python, состоящего из объектов-точек `wx.Point`. Отличается от `DrawLines()` в том, что конечная точка соединяется с первой точкой, а результирующая форма закрашивается текущей кистью. Опциональные параметры задают смещение по x и y, а также стиль закрашивания.
`DrawRectangle(x, y, width, height)`	Рисует прямоугольник, имеющий левый верхний угол в точке (x, y), ширину width и высоту height. Прямоугольник закрашивается.
`DrawText(text, x, y)`	Используя текущий шрифт, рисует строку текста в точке (x, y). Смежные функции: `DrawRotatedText()` и `GetTextExtent()`. Текстовые элементы имеют независимые свойства цвета текста и цвета фона.

`FloodFill(x, y, color, style)`	Выполняет закрашивание области, начиная в точке (x, y) и используя цвет текущей кисти. Параметр `style` опциональный. Его значение по умолчанию `wx.FLOOD_SURFACE` указывает, что параметр `color` определяет поверхность закрашивания, которое прекращается, когда обнаруживается любой другой цвет. Другое значение, `wx.FLOOD_BORDER`, определяет, что цвет является границей закрашиваемой формы, и при обнаружения этого цвета закрашивание прекращается.
`GetBackground()` `SetBackground(brush)`	Фоновая кисть является объектом `wx.Brush` и используется при вызове метода `Clear()`.
`GetBrush()` `SetBrush(brush)`	Эта кисть является объектом `wx.Brush` и используется для закрашивания любых фигур, которые рисуются в контексте устройства.
`GetFont()` `SetFont(font)`	Это шрифт, являющийся объектом `wx.Font` и используется для всех операций рисования текста.
`GetPen()` `SetPen(pen)`	Это перо является объектом `wx.Pen` и используется во всех операциях рисования, где рисуются линии.
`GetPixel(x, y)`	Возвращает объект `wx.Colour` для пикселя в точке (x, y).
`GetSize()` `GetSizeTuple()`	Возвращает размер пикселя в контексте устройства в виде объекта `wx.Size` или как кортеж Python.

Это не полный список. В целях упрощения, были пропущены некоторые менее примечательные методы рисования, какими являются функции текстовой обработки и пиксельной манипуляции. Эти методы будут описаны в главе 12.

6.2 Добавление оконных элементов оформления

При рисовании на экране существенная часть программы рисования далека от тех вещей, которые придают Вашему приложению изящный вид. В этом подразделе мы обсудим общие элементы оформления окна: панель состояния (status bar), полосу меню (menubar) и инструментальную панель (toolbar). К тому же, в главе 10 мы обсудим эти возможности более подробно.

6.2.1 Как добавить и доработать панель состояния?

В wxPython Вы можете добавить и разместить панель состояния внизу фрейма, вызывая метод фрейма `CreateStatusBar()`. Панель состояния автоматически изменяет свои размеры вместе с родительским фреймом. По умолчанию панель

состояния является экземпляром класса `wx.StatusBar`. Для создания пользовательского подкласса панели состояния подключите его к Вашему фрейму при помощи метода `SetStatusBar()`, передавая в качестве аргумента экземпляр Вашего нового класса.

Для отображения в Вашей панели состояния отдельного фрагмента текста, Вы можете использовать метод `SetStatusText()` класса `wx.StatusBar`. Листинг 6.2 расширяет приведенный в листинге 6.1 класс `SketchFrame`, для отображения в панели состояния текущей позиции указателя мыши.

Листинг 6.2 Добавление к фрейму простой панели состояния

```
import wx
from example1 import SketchWindow

class SketchFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, "Sketch Frame",
                size=(800,600))
        self.sketch = SketchWindow(self, -1)
        self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
        self.statusbar = self.CreateStatusBar()

    def OnSketchMotion(self, event):
        self.statusbar.SetStatusText(str(event.GetPositionTuple()))
        event.Skip()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = SketchFrame(None)
    frame.Show(True)
    app.MainLoop()
```

Мы подключили панель состояния, при этом фрейм также может обрабатывать событие `wx.EVT_MOTION` нашего окна рисования. Обработчик события устанавливает текст панели состояния на основании предоставляемых событием данных. Затем он вызывает метод `Skip()`, обеспечивающий вызов другого метода `OnMotion()`, в противном случае линия не будет нарисована.

Помещая на панель состояния дополнительные объекты, Вы можете сделать её похожей на любой другой виджет. Если Вы захотите вывести в панели состояния несколько текстовых элементов, можете создать множество текстовых полей. Чтобы использовать эту возможность, вызовите метод `SetFieldsCount()` с необходимым Вам количеством полей; значение по умолчанию – одно поле. После это, используйте `SetStatusText()` как в ранее приведенном примере, но со вторым аргументом, задающим номер устанавливаемого методом поля. Нумерация полей начинается с нуля. Если Вы не укажите номер поля, то по умолчанию устанавливается поле с номером 0. Вот почему работает предыдущий пример, где мы так и поступили.

По умолчанию, все поля имеют равную ширину. Тем не менее, это не всегда удобно. Для того чтобы отрегулировать размеры текстовых полей, wxPython предоставляет метод `SetStatusWidth()`. Этот метод принимает список целых значений, определяющих длину полей панели состояния. Данный список целых значений используется, чтобы надлежащим образом вычислять ширину полей. Если целое положительно, оно определяет фиксированную ширину поля. Если Вы хотите, чтобы ширина поля изменялась с фреймом, укажите отрицательное целое. Абсолютная величина отрицательного целого указывает относительный размер поля, выраженный количеством отводимых полю частей в общей ширине панели. Например, вызов `statusbar.SetStatusWidth([-1, -2, -3])` отводит для самого правого поля половину ширины (3/6 части), центральная область получает треть ширины (2/6 части), а крайнее левое поле получает шестую часть ширины (1/6 часть). Результат показан на рисунке 6.2.

Рисунок 6.2 Пример панели состояния с полями в 1/6, 2/3 и 1/2 от общей ширины

Листинг 6.3 добавляет поддержку двух дополнительных полей панели состояния, одно из который показывает количество точек в текущей проведенной линии, а другое показывает количество линий в текущем чертеже. Этот листинг воспроизводит панель состояния, показанную на рисунке 6.2.

Листинг 6.3 Поддержка множества полей состояния

```
import wx
from example1 import SketchWindow

class SketchFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, "Sketch Frame",
                size=(800,600))
        self.sketch = SketchWindow(self, -1)
        self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
        self.statusbar = self.CreateStatusBar()
        self.statusbar.SetFieldsCount(3)
        self.statusbar.SetStatusWidths([-1, -2, -3])

    def OnSketchMotion(self, event):
        self.statusbar.SetStatusText("Pos: %s" %
                str(event.GetPositionTuple()), 0)
        self.statusbar.SetStatusText("Current Pts: %s" %
                len(self.sketch.curLine), 1)
        self.statusbar.SetStatusText("Line Count: %s" %
                len(self.sketch.lines), 2)
        event.Skip()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = SketchFrame(None)
    frame.Show(True)
```

```
app.MainLoop()
```

Класс `wx.StatusBar` позволяет Вам рассматривать поля панели состояния как стек типа «последний вошёл - первый вышел» (LIFO). Методы `PushStatusText()` и `PopStatusText()` позволять после временного отображения нового текста вернуться к предыдущему тексту поля панели состояния (правда, в демонстрационном приложении данной главы этого не требуется). Оба этих метода принимают дополнительный номер поля, поэтому они могут использоваться и в случае панели с множеством полей.

Таблица 6.3 приводит наиболее употребительные методы класса `wx.StatusBar`.

Таблица 6.3 Методы класса `wx.StatusBar`

Метод	Описание
`GetFieldsCount()` `SetFieldsCount(count)`	Свойство, определяющее количество полей в панели состояния.
`GetStatusText(field=0)` `SetStatusText(text, field=0)`	Свойство, определяющее выводимый в указанном поле текст. Индекс 0 (значение по умолчанию) указывает крайнее левое поле.
`PopStatusText(field=0)`	Извлекает текст из стека указанного поля панели состояния, замещая текст этого поля извлеченным значением.
`PushStatusText(text, field=0)`	Замещает вид указанного поля панели состояния данным текстом и помещает это новое значение в вершину стека данного поля.
`SetStatusWidths(widths)`	Берёт список целых значений и определяет ширину полей панели состояния. Положительное число указывает фиксированную ширину в пикселях, а отрицательное число указывает динамическую часть ширины пропорциональную абсолютной величине числа.

В главе 10 будут предоставлены дополнительные сведения о панелях состояния. А сейчас мы обсудим меню.

6.2.2 *Как добавить подменю или меню с флажками?*

В этом подразделе, мы представим два распространенных приёма для меню - это подменю и меню в виде флажков (checked menu) или переключателей (radio menu). Подменю – это меню, которое доступно в одном из меню верхнего уровня. Меню-флажок или меню-переключатель – это группа пунктов меню, которые ведут себя подобно группе флажков или переключателей. Рисунок 6.3 показывает строку меню, включающее подменю с пунктами в виде переключателей.

Для создания подменю, сформируйте его, как и любое другое меню, и добавьте в родительское меню при помощи метода `wx.Меню.AppendMenu()`.

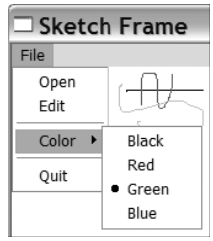

Рисунок 6.3 Меню и подменю с пунктами-переключателями

Пункты меню с украшениями в виде флажков или переключателей могут также создаваться при помощи методов `AppendCheckItem()` и `AppendRadioItem()` класса `wx.Menu`, или передавая атрибутом типа в конструкторе `wx.MenuItem` одно из следующих значений: `wx.ITEM_NORMAL`, `wx.ITEM_CHECKBOX` или `wx.ITEM_RADIO`. Пункт меню в виде флажка отображает отметку, которая автоматически включается или выключается при выборе этого пункта; Вам не нужно вручную управлять этим процессом. Начальное значение пункта меню в виде флажка - выключено. Пункты меню в виде переключателей сгруппированы. Предполагается, что последовательные пункты-переключатели будут частью одной и той же группы (разделитель меню разрывает группу). По умолчанию, отмечен самый верхний элемент группы, а при выборе любого элемента группы отметка автоматически передаётся на соответствующий выбранный пункт. Для того чтобы программно отметить пункт меню, используйте метод `Check(id, bool)` класса `wx.Menu`, где `id` – wxPython-идентификатор того пункта меню, который нужно изменить, а `bool` определяет состояние этого пункта.

Листинг 6.4 добавляет меню к фрейму приложения. Функциональное назначение этого кода – развитие предшествующего рефакторизованного кода из листинга 5.5. Для создания подменю здесь усовершенствован формат данных, а код создания при необходимости создает подменю рекурсивно. Также добавлена поддержка меню с переключателями и флажками.

Листинг 6.4 Поддержка в приложении меню

```
import wx
from example1 import SketchWindow

class SketchFrame(wx.Frame):
    def __init__(self, parent):
        wx.Frame.__init__(self, parent, -1, "Sketch Frame",
                size=(800,600))
        self.sketch = SketchWindow(self, -1)
        self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
        self.initStatusBar()
        self.createMenuBar()

    def initStatusBar(self):
        self.statusbar = self.CreateStatusBar()
        self.statusbar.SetFieldsCount(3)
        self.statusbar.SetStatusWidths([-1, -2, -3])

    def OnSketchMotion(self, event):
        self.statusbar.SetStatusText("Pos: %s" %
                str(event.GetPositionTuple()), 0)
```

1 **Небольшой рефакторинг**

```
    self.statusbar.SetStatusText("Current Pts: %s" %
            len(self.sketch.curLine), 1)
    self.statusbar.SetStatusText("Line Count: %s" %
            len(self.sketch.lines), 2)
    event.Skip()

def menuData(self):
    return [("&File", (
                ("&New", "New Sketch file", self.OnNew),
                ("&Open", "Open sketch file", self.OnOpen),
                ("&Save", "Save sketch file", self.OnSave),
                ("", "", ""),
                ("&Color", (
                    ("&Black", "", self.OnColor,
                       wx.ITEM_RADIO),
                    ("&Red", "", self.OnColor,
                       wx.ITEM_RADIO),
                    ("&Green", "", self.OnColor,
                       wx.ITEM_RADIO),
                    ("&Blue", "", self.OnColor,
                       wx.ITEM_RADIO))),
                ("", "", ""),
              ("&Quit", "Quit", self.OnCloseWindow)))]

def createMenuBar(self):
    menuBar = wx.MenuBar()
    for eachMenuData in self.menuData():
        menuLabel = eachMenuData[0]
        menuItems = eachMenuData[1]
        menuBar.Append(self.createMenu(menuItems), menuLabel)
    self.SetMenuBar(menuBar)

def createMenu(self, menuData):
    menu = wx.Menu()
    for eachItem in menuData:
        if len(eachItem) == 2:
            label = eachItem[0]
            subMenu = self.createMenu(eachItem[1])
            menu.AppendMenu(wx.NewId(), label, subMenu)
        else:
            self.createMenuItem(menu, *eachItem)
    return menu

def createMenuItem(self, menu, label, status, handler,
                   kind=wx.ITEM_NORMAL):
    if not label:
        menu.AppendSeparator()
        return
    menuItem = menu.Append(-1, label, status, kind)
    self.Bind(wx.EVT_MENU, handler, menuItem)

def OnNew(self, event): pass
def OnOpen(self, event): pass
def OnSave(self, event): pass
```

Идентификация данных меню

2

Создание подменю

3

4

Создание элементов меню с типом

```
    def OnColor(self, event):
        menubar = self.GetMenuBar()
        itemId = event.GetId()
        item = menubar.FindItemById(itemId)
        color = item.GetLabel()
        self.sketch.SetColor(color)

    def OnCloseWindow(self, event):
        self.Destroy()

if __name__ == '__main__':
    app = wx.PySimpleApp()
    frame = SketchFrame(None)
    frame.Show(True)
    app.MainLoop()
```

5 Обработка изменения цвета

1. Теперь метод `__init__` более функционален и мы инкапсулировали инициализацию панели состояния в отдельный метод.
2. Здесь определяется формат данных меню в виде (метка, (пункты)), где каждый пункт – это или список вида (метка, текст панели состояния, обработчик, опциональный тип) или меню с меткой и пунктами. Для определения того, какие подпункты данных являются меню или пунктами меню, помните, что меню содержит 2, а пункты 3 или 4 элемента данных. В промышленной системе, где данные имеют более сложную структуру, я рекомендую использовать XML или какой-нибудь другой внешний формат.
3. Когда данные имеет длину 2, что означает подменю, производится разделение меню тем же способом, как и для меню верхнего уровня, при этом рекурсивно вызывается добавляющий подменю метод `createMenu`.
4. При такой реализации меню проще добавить в конструктор `wx.MenuItem` параметр типа, чем использовать специальные методы `wx.Menu`.
5. Для обработки изменения цвета у всех пунктов меню установлен метод `OnColor`, при этом не требуется установки отдельных обработчиков для каждого пункта. В этом случае код получает идентификатор (`id`) пункта из события и использует метод `FindItemById()` для получения соответствующего пункта меню (обратите внимание, что от нас не требуется поддерживать отдельную хеш-таблицу идентификаторов пунктов – мы используем полосу меню как структуру данных). Этот метод полагает, что метка пункта меню является наименованием цвета wxPython и он передает эту метку окну рисования, где изменяется перо.

6.2.3 Как добавить панель инструментов?

Рисунок 6.4 Типичная панель инструментов с регулярными и переключающими кнопками

Строка меню и панель инструментов часто тесно взаимосвязаны, причем бо́льшая часть или даже вся функциональность панели инструментов доступна и через пункты меню. В wxPython это сходство расширено кнопками панели инструментов, генерирующими при нажатии события `wx.EVT_MENU`, что облегчает использование одних и тех же методов, как для обработки выбора пунктов меню, так и щелчков на панели инструментов. Панель инструментов wxPython – это экземпляр класса `wx.ToolBar`, и как мы увидели в главе 2, она может быть создана при помощи метода фрейма `CreateToolBar()`. Подобно панели состояния, инструментальная панель автоматически меняет размеры вместе с родительским фреймом. Инструментальная панель подобна другим окнам wxPython в том, что она может иметь произвольные дочерние окна. Панели инструментов содержат также методы для создания инструментальных кнопок. Рисунок 6.4 показывает часть окна с панелью инструментов, которая дублирует функциональность созданного нами меню.

Как и в коде меню, цветные битовые изображения представляют собой переключатели, и переключение одного из них приводит к изменению выделения. В листинге 6.5 мы не повторяем код меню, а включили новые и измененные методы `SketchFrame`.

Листинг 6.5 Добавление в приложение панели инструментов

```
def __init__(self, parent):
    wx.Frame.__init__(self, parent, -1, "Sketch Frame",
            size=(800,600))
    self.sketch = SketchWindow(self, -1)
    self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
    self.initStatusBar()
    self.createMenuBar()
    self.createToolBar()

def createToolBar(self):
    toolbar = self.CreateToolBar()
    for each in self.toolbarData():
        self.createSimpleTool(toolbar, *each)
    toolbar.AddSeparator()
    for each in self.toolbarColorData():
        self.createColorTool(toolbar, each)
    toolbar.Realize()
```

1 **Создание панели инструментов**

2 **Реализация панели инструментов**

```
def createSimpleTool(self, toolbar, label, filename,
        help, handler):
    if not label:
        toolbar.AddSeparator()
        return
    bmp = wx.Image(filename,
            wx.BITMAP_TYPE_BMP).ConvertToBitmap()
    tool = toolbar.AddSimpleTool(-1, bmp, label, help)
    self.Bind(wx.EVT_MENU, handler, tool)

def toolbarData(self):
    return (("New", "new.bmp", "Create new sketch",
                self.OnNew),
            ("", "", "", ""),
            ("Open", "open.bmp", "Open existing sketch",
                self.OnOpen),
            ("Save", "save.bmp", "Save existing sketch",
                self.OnSave))

def createColorTool(self, toolbar, color):
    bmp = self.MakeBitmap(color)
    newId = wx.NewId()
    tool = toolbar.AddRadioTool(-1, bmp, shortHelp=color)
    self.Bind(wx.EVT_MENU, self.OnColor, tool)

def MakeBitmap(self, color):
    bmp = wx.EmptyBitmap(16, 15)
    dc = wx.MemoryDC()
    dc.SelectObject(bmp)
    dc.SetBackground(wx.Brush(color))
    dc.Clear()
    dc.SelectObject(wx.NullBitmap)
    return bmp

def toolbarColorData(self):
    return ("Black", "Red", "Green", "Blue")

def OnColor(self, event):
    menubar = self.GetMenuBar()
    itemId = event.GetId()
    item = menubar.FindItemById(itemId)
    if not item:
        toolbar = self.GetToolBar()
        item = toolbar.FindById(itemId)
        color = item.GetShortHelp()
    else:
        color = item.GetLabel()
    self.sketch.SetColor(color)
```

3 **Создание простых инструментов**

Создание инструментов выбора цвета 4

5 **Создание сплошного битового изображения**

6 **Изменение цвета при щелчке на панели инструментов**

1 Код для панели инструментов аналогичен по настройке коду для меню в том, что он также управляется на основе данных. Тем не менее, в этом случае, мы установили другие циклы для обычных кнопок и для кнопок-переключателей, поскольку они не внедрены в панель инструментов.

2. Метод `Realize()` размещает объекты внутри инструментальной панели. Он должен быть вызван до отображения панели, и повторно вызывается, если в панель добавляются или удаляются любые инструменты.

3. Этот метод похож на метод создания пунктов меню. Основное отличие в том, что для инструментов панели требуются битовые изображения. В данном случае, мы разместили три основных битовых изображения в том же каталоге, где размещается код примера. В конце метода мы назначаем тоже событие `wx.EVT_MENU`, которое использовано для пунктов меню. Для расшифровки метода `AddTool`, который обеспечивает более специфические функции инструментов, используйте таблицу 6.4.

4. Инструментальные кнопки для выбора цвета создаются аналогично простым инструментам, просто панели разным способом сообщается о том, что они являются кнопками-переключателями. Сплошные битовые изображения созданы методом `MakeBitmap()`.

5. Этот метод создает сплошное (solid) битовое изображение соответствующего размера, создавая обычное битовое изображение `wx.EmptyBitmap`, подключая к нему `wx.MemoryDC`, и очищая битовое изображение нужным цветом при помощи фоновой кисти.

6. Небольшое дополнение к методу `OnColor()` ищет в панели соответствующий инструмент и устанавливает необходимый цвет. Однако в коде имеется одна проблема - изменение цвета через пункт меню не изменяет состояние переключателей панели инструментов, и наоборот.

Панели инструментов имеют возможность гибкого управления событиями, которую не имеют пункты меню. Они могут сгенерировать событие типа `wx.EVT_TOOL_RCLICKED`, когда на инструменте нажимается правая кнопка мыши. К тому же, панели инструментов имеют несколько различных стилей, которые передаются как битовые изображения в качестве аргументов в `CreateToolBar()`. Таблица 6.4 перечисляет некоторые стили панели инструментов.

Таблица 6.4 Стили класса `wx.ToolBar`

Стиль	Описание
`wx.TB_3DBUTTONS`	Представляет инструменты в виде 3D
`wx.TB_HORIZONTAL`	Этот стиль по умолчанию размещает панель инструментов горизонтально
`wx.TB_NOICONS`	Не отображает битовые изображения для каждого инструмента
`wx.TB_TEXT`	Панель инструментов будет показывать короткий вспомогательный текст вместе со встроенными битовыми изображениями

`wx.TB_VERTICAL`	Размещает панель инструментов вертикально

Панели инструментов несколько сложнее панелей состояния. Таблица 6.5 отображает некоторые широко используемые методы панелей инструментов.

Таблица 6.5 Методы общего назначения класса `wx.ToolBar`

Функция	Описание
`AddControl(control)`	Добавляет в панель инструментов произвольный управляющий виджет wxPython. Также смотрите связанный метод `InsertControl()`.
`AddSeparator()`	Устанавливает между инструментами пустое пространство.
`AddSimpleTool(id, bitmap, shortHelpString="", kind=wx.ITEM_NORMAL)` `AddTool(id, bitmap, bitmap2=wx.NullBitmap, kind=wx.ITEM_NORMAL, shortHelpString="", longHelpString="", clientData=None)`	Добавляет в панель инструментов простую инструментальную кнопку с данным битовым изображением. В качестве подсказки (tooltip) отображается `shortHelpString`. Параметр `kind` может иметь значение `wx.ITEM_NORMAL`, `wx.ITEM_CHECKBOX` или `wx.ITEM_RADIO`. Дополнительные параметры для простых инструментов: `bitmap2` отображается, когда инструмент нажат; `longHelpString` отображается на панели состояния, когда указатель находится на инструменте, а `clientData` может быть использован, чтобы ассоциировать с инструментом произвольную часть данных. Имеется связанный метод `InsertTool()`.
`AddCheckTool(...)`	Добавляет инструмент в виде флажка, с теми же параметрами, как и у `AddTool()`.
`AddRadioTool(...)`	Добавляет инструмент в виде переключателя, с теми же параметрами, как и у `AddTool()`. Непрерывная последовательность переключателей объединяется в группу.
`DeleteTool(toolId)` `DeleteToolByPosition(x, y)`	Удаляет инструмент с данным идентификатором `toolId`, или тот, который отображен в данной точке.
`FindControl(toolId)` `FindToolForPosition(x, y)`	Находит и возвращает инструмент с данным идентификатором `toolId`, или тот, который отображен в данной точке.
`ToggleTool(toolId, toggle)`	Если инструмент с указанным идентификатором `toolId` является переключателем или флажком, этот метод устанавливает переключатель данного инструмента, полагаясь на булевое значение аргумента `toggle`.

В следующем разделе мы покажем Вам, как использовать стандартные диалоги, для того чтобы получать информацию от пользователя. В большинстве операционных систем Вы можете использовать стандартные диалоги, чтобы обеспечивать Вашего пользователя знакомым интерфейсом при решении общих задач, таких как, например, выбор файла.

6.3 Получение стандартной информации

Ваше приложение часто нуждается в получении от пользователя основной информации, что обычно выполняется посредством диалоговых окон. В этом разделе мы обсудим использование стандартных диалогов выбора файлов и цвета как стандарта информационного взаимодействия пользователя.

6.3.1 Как использовать стандартные файловые диалоги?

Большинство приложений с GUI должны сохранять и загружать данные того или иного типа, поэтому Вам и Вашим пользователям, желательно иметь единый и согласованный механизм выбора файлов. К счастью, wxPython реализует с этой целью стандартный диалог `wx.FileDialog`, который может включаться в Ваши приложения. Под MS Windows, этот класс реализован на основе стандартного файлового диалога Windows. В системе X Window он выглядит подобно обычному пользовательскому диалогу. Рисунок 6.5 показывает файловый диалог нашего приложения для зарисовок.

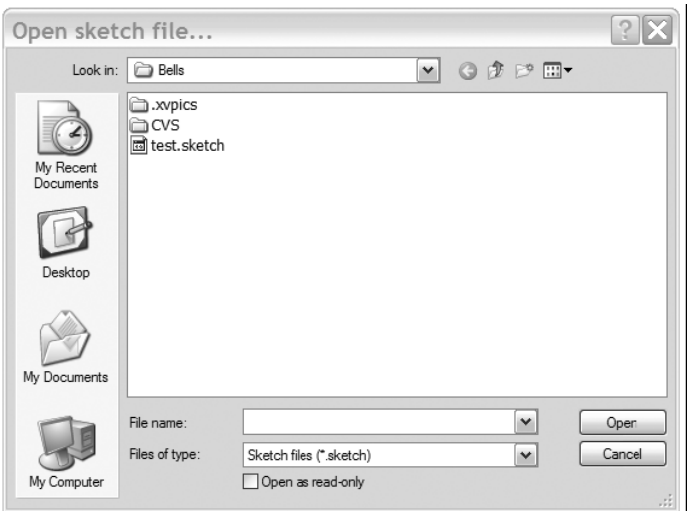

Рисунок 6.5 Стандартный файловый диалог Windows

Важнейший метод при использовании `wx.FileDialog` – это конструктор. Он имеет следующий вид:

```
wx.FileDialog(parent, message="Choose a file", defaultDir="",
        defaultFile="", wildcard="*.*", style=0)
```

Таблица 6.6 описывает параметры данного конструктора.

Таблица 6.6 Параметры конструктора `wx.FileDialog`

Параметр	Описание
`parent`	Родительское окно для диалога, или значение None, если нет родительского окна.
`message`	Сообщение, отображаемое в полосе заголовка диалога.
`defaultDir`	Каталог, с которого диалог должен стартовать. Если пусто, диалог стартует в текущем рабочем каталоге.
`defaultFile`	Выбираемый при открытии диалога файл. Если пусто, никакой файл не выбирается.
`wildcard`	Опции, определяющие шаблонный фильтр, который позволяют пользователю ограничивать показ файлов выбранного типа. Формат `<вид>|<шаблон>` может повторяться многократно, что предоставляет пользователю альтернативные варианты; например, "Файлы схем (*.sketch)|*.sketch|Все файлы (*.*)|*.*"
`style`	Стилевая битовая маска. Стили указаны в таблице 6.7.

Таблица 6.7 содержит опции стилевой битовой маски.

Таблица 6.7 Стилевые опции `wx.FileDialog`

Стиль	Описание
`wx.CHANGE_DIR`	После того, как пользователь выберет файл, текущим рабочим каталогом станет данный каталог.
`wx.MULTIPLE`	Этот стиль применяется только для диалога открытия и позволяет пользователю выбирать множество файлов.
`wx.OPEN`	Этот стиль используется для открытия файла.
`wx.OVERWRITE_PROMPT`	Этот стиль применяется только для диалога сохранения и указывает, что необходимо выдавать сообщение для подтверждения перезаписи существующего файла.
`wx.SAVE`	Этот стиль используется для сохранения файла.

Для того чтобы использовать файловый диалог, вызовите для экземпляра диалога метод `ShowModal()`. Этот метод возвращает значение `wx.ID_OK` или `wx.ID_CANCEL`, в зависимости от того, какую кнопку для закрытия диалога щелкнул пользователь. После выбора используйте методы `GetFilename()`, `GetDirectory()` или `GetPath()`, чтобы извлечь данные. Впоследствии неплохо было бы уничтожить диалог методом `Destroy()`.

Листинг 6.6 показывает необходимые модификации в `SketchFrame` для обеспечения функций сохранения и загрузки. Эти изменения требуют также импортирования стандартных модулей `cPickle` и `os`. Мы будем использовать `cPickle` для преобразования списка данных окна чертежа в формат, который может быть записан и прочитан из файла.

Листинг 6.6 Методы сохранения и загрузки для `SketchFrame`

```
def __init__(self, parent):
    self.title = "Sketch Frame"
    wx.Frame.__init__(self, parent, -1, self.title,
            size=(800,600))
    self.filename = ""
    self.sketch = SketchWindow(self, -1)
    self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
    self.initStatusBar()
    self.createMenuBar()
    self.createToolBar()

def SaveFile(self):
    if self.filename:
        data = self.sketch.GetLinesData()
        f = open(self.filename, 'w')
        cPickle.dump(data, f)
        f.close()

def ReadFile(self):
    if self.filename:
        try:
            f = open(self.filename, 'r')
            data = cPickle.load(f)
            f.close()
            self.sketch.SetLinesData(data)
        except cPickle.UnpicklingError:
            wx.MessageBox("%s is not a sketch file."
                    % self.filename, "oops!",
                    style=wx.OK|wx.ICON_EXCLAMATION)

wildcard = "Sketch files (*.sketch)|*.sketch|All files (*.*)|*.*"

def OnOpen(self, event):
    dlg = wx.FileDialog(self, "Open sketch file...",
            os.getcwd(), style=wx.OPEN,
            wildcard=self.wildcard)
    if dlg.ShowModal() == wx.ID_OK:
        self.filename = dlg.GetPath()
        self.ReadFile()
        self.SetTitle(self.title + ' -- ' + self.filename)
    dlg.Destroy()

def OnSave(self, event):
    if not self.filename:
        self.OnSaveAs(event)
```

1 **Сохранение файла**

2 **Чтение файла**

3 **Вывод диалога открытия**

4 **Сохранение файла**

```
        else:
            self.SaveFile()

    def OnSaveAs(self, event):
        dlg = wx.FileDialog(self, "Save sketch as...",
                os.getcwd(),
                style=wx.SAVE | wx.OVERWRITE_PROMPT,
                wildcard=self.wildcard)
        if dlg.ShowModal() == wx.ID_OK:
            filename = dlg.GetPath()
            if not os.path.splitext(filename)[1]:
                filename = filename + '.sketch'
            self.filename = filename
            self.SaveFile()
            self.SetTitle(self.title + ' -- ' +
                    self.filename)
        dlg.Destroy()
```

5 Вывод диалога сохранения

6 Поддержка расширения файла

1. Этот метод с использованием модуля `cPickle` записывает на диск файл данных, указанный в `filename`.
2. Этот метод читает файл, используя `cPickle`. Если файл имеет неверный тип, будет выдано предупредительное сообщение.
3. Метод `OnOpen()` создает открываемый в текущем каталоге диалог со стилем `wx.OPEN`. Строка шаблона `wildcard` на предыдущей строке позволяет ограничить выбор пользователя файлами `.sketch`. Если пользователь щелкнет кнопку OK, вызывается метод `ReadFile()` с выбранным в диалоге путем к файлу (полное имя файла).
4. Если имя файла для текущих данных уже выбрано, мы сохраняем файл, в противном случае, мы обрабатываем такую ситуацию способом «сохранить как» и открываем диалог сохранения.
5. Метод `OnSave()` создает файловый диалог со стилем `wx.SAVE`.
6. Эта строка гарантирует, что набранное без расширения имя файла получит расширение `.sketch`.

В следующем разделе мы обсудим использование диалога выбора цвета.

6.3.2 *Как использовать стандартный диалог выбора цвета?*

Целесообразно предоставить пользователю возможность выбирать произвольный цвет рисования в диалоге. Для этой цели мы можем использовать стандартный диалог wxPython `wx.ColourDialog`. Использование этого диалога подобно файловому диалогу. Конструктору передается только родитель и дополнительный атрибут данных. Атрибут данных является экземпляром `wx.ColourData`, хранящим некоторые связанные с диалогом данные, например, выбранный пользователем

цвет и список пользовательских цветов. Применение атрибута данных позволяет Вам хранить пользовательские цвета в процессе их использования.

Использование диалога выбора цвета в нашем приложении требует дополнительный пункт меню, а также выразительный и простой метод-обработчик. Листинг 6.7 показывает дополнения программного кода.

Листинг 6.7 Изменения в `SketchFrame` для вывода диалога выбора цвета

```
    def menuData(self):
        return [("&File", (
                    ("&New", "New Sketch file", self.OnNew),
                    ("&Open", "Open sketch file", self.OnOpen),
                    ("&Save", "Save sketch file", self.OnSave),
                    ("", "", ""),
                    ("&Color", (
                        ("&Black", "", self.OnColor,
                                wx.ITEM_RADIO),
                        ("&Red", "", self.OnColor,
                                wx.ITEM_RADIO),
                        ("&Green", "", self.OnColor,
                                wx.ITEM_RADIO),
                        ("&Blue", "", self.OnColor,
                                wx.ITEM_RADIO),
                        ("&Other...", "", self.OnOtherColor,
                                wx.ITEM_RADIO))),
                    ("", "", ""),
                    ("&Quit", "Quit", self.OnCloseWindow")))]

    def OnOtherColor(self, event):
        dlg = wx.ColourDialog(self)
        dlg.GetColourData().SetChooseFull(True)
        if dlg.ShowModal() == wx.ID_OK:
            self.sketch.SetColor(dlg.GetColourData().GetColour())
        dlg.Destroy()
```

Создание объекта цветовых данных

Установка введенного пользователем цвета

Мы сделали две вещи с диалогом выбора цвета, которые не очевидны при первом взгляде. Метод `SetChooseFull()` экземпляра цветовых данных указывает диалогу отображаться с полной палитрой, включая информацию о пользовательских цветах. После закрытия диалога мы снова обращаемся к цветовым данным для получения цвета. Цветовые данные возвращаются в виде экземпляра `wx.Color` и пригодны для передачи в объект зарисовки для установки цвета.

6.4 Улучшение внешнего вида приложения

В этом разделе мы обсудим вопросы, касающиеся того, как придать Вашему приложению законченный внешний вид. Круг рассматриваемых вопросов включает как существенные вопросы, например то, как Вам размещать элементы, чтобы пользователь мог изменять размеры окна, так и более тривиальные,

например, как отобразить диалог About. Подробнее эти темы рассматриваются в части 2.

6.4.1 Как располагать виджеты?

Один из способов состоит в том, чтобы размещать Ваши виджеты в приложении wxPython, явно определяя позицию и размер каждого виджета при его создании. Хотя этот метод довольно прост, тем не менее, у него есть ряд недостатков. Прежде всего, из-за того, что размеры виджета и размеры шрифта по умолчанию отличаются, очень трудно выполнять точное позиционирование на всех системах. Кроме того, Вы должны явно изменять позицию каждого виджета всякий раз, когда пользователь изменяет размеры родительского окна. Это может вызвать реальную проблему для корректной реализации.

К счастью, имеется лучший способ. Это механизм размещения wxPython, именуемый *координатором* (sizer), его идея аналогична менеджерам размещения в Java AWT и других пакетах разработчика интерфейса. Каждый отдельный координатор управляет размером и позицией своего окна, основываясь на ряде критериев. Координатор является частью контейнерного окна (обычно это `wx.Panel`). Создающиеся внутри родителя дочерние окна должны помещаться в координатор, а он в свою очередь управляет размером и позицией каждого виджета.

Создание координатора

Для создания координатора:

1. Создайте панель или контейнер, размеры которого Вы хотите изменять автоматически.
2. Создайте координатор.
3. Создайте, как и обычно, Ваши дочерние окна.
4. Добавьте каждое дочернее окно в координатор при помощи метода `Add()`. При этом дочерние окна помещаются в родительский контейнер. При добавлении окна, предоставьте координатору дополнительную информацию, включающую количество свободного места вокруг окна, способ выравнивания окна в пределах управляемого координатором пространства и способ расширения окна при изменении размеров контейнерного окна.
5. Координаторы могут вкладываться, т.е. наравне с оконными объектами Вы можете добавлять в родительский координатор другие координаторы. Вы можете также задать определенное количество свободного места в качестве разделителя.
6. Вызовите метод контейнера `SetSizer(sizer)`.

Таблица 6.8 включает наиболее употребительные координаторы wxPython. За более полным описанием каждого конкретного координатора обращайтесь к главе 11.

Таблица 6.8 Наиболее употребительные координаторы wxPython

Координатор	Описание
`wx.BoxSizer`	Размещает вложенные элементы в линию. Для создания сложных видов размещения координатор `wx.BoxSizer` может быть горизонтально или вертикально ориентированным и может содержать вложенные координаторы любой ориентации. Параметры, передаваемые координатору при добавлении элементов, управляют тем, как дочерние элементы реагируют на изменение размеров вдоль основной или перпендикулярной оси контейнера.
`wx.GridSizer`	Фиксированная двумерная сетка, где все элементы имеют одинаковый размер, соответствующий размеру наибольшего элемента в координаторе. При создании сеточного координатора, Вы фиксируете или количество столбцов или количество рядов. Элементы добавляются слева направо до тех пор, пока ряд не будет заполнен, затем начинается следующий ряд.
`wx.FlexGridSizer`	Фиксированная двумерная сетка, которая отличается от `wx.GridSizer` тем, что размер каждого ряда и столбца устанавливается отдельно на основании наибольшего элемента в этом ряде или столбце.
`wx.GridBagSizer`	Двумерная сетка, базирующаяся в `wx.FlexGridSizer`. Позволяет разместить все элементы в определенных точках сетки, а также позволяет распределять составные позиции сетки.
`wx.StaticBoxSizer`	Аналог координатора `wx.BoxSizer`, дополненный границей вокруг контейнера и опциональным заголовком.

Использование координатора

Чтобы продемонстрировать использование координатора, мы добавим в наше приложение для зарисовок панель управления. Панель управления содержит кнопки для установки цвета и толщины линии. Этот пример использует экземпляры `wx.GridSizer` (для кнопок) и `wx.BoxSizer` (для остальных элементов). Рисунок 6.6 показывает наше приложение с панелью, иллюстрируя на практике вид сеточного и линейного размещения.

Рисунок 6.6 Приложение Sketch с автоматически размещенной панелью управления

Листинг 6.8 показывает изменения в приложении Sketch, которые необходимо выполнить для реализации панели управления. Обсуждение этого раздела фокусируется на реализации координатора.

Листинг 6.8 Внесение координатора в `SketchFrame`

```
    def __init__(self, parent):
        self.title = "Sketch Frame"
        wx.Frame.__init__(self, parent, -1, self.title,
                size=(800,600))
        self.filename = ""
        self.sketch = SketchWindow(self, -1)
        self.sketch.Bind(wx.EVT_MOTION, self.OnSketchMotion)
        self.initStatusBar()
        self.createMenuBar()
        self.createToolBar()
        self.createPanel()

    def createPanel(self):
        controlPanel = ControlPanel(self, -1, self.sketch)
        box = wx.BoxSizer(wx.HORIZONTAL)
        box.Add(controlPanel, 0, wx.EXPAND)
        box.Add(self.sketch, 1, wx.EXPAND)
        self.SetSizer(box)
```

В листинге 6.8 метод `createPanel()` создает экземпляр `ControlPanel` (описанный в следующем листинге) и компонует линейный координатор. Единственный параметр конструктора для `wx.BoxSizer` - ориентация, которая может иметь значение `wx.HORIZONTAL` или `wx.VERTICAL`. Затем новая панель управления и созданный ранее `SketchWindow` добавляются в координатор при помощи метода `Add()`. Первый аргумент является добавляемым в координатор объектом. Второй аргумент `wx.BoxSizer` используется в качестве коэффициента растяжения, и определяет, как координатору изменять размеры своих вложенных элементов при изменении его собственного размера. В случае горизонтального координатора, коэффициент растяжения определяет, как изменяется горизонтальный размер каждого вложенного элемента (вертикальное растяжение выполняется линейным координатором с помощью флажков в третьем аргументе).

Если коэффициент растяжения равен нулю, объект не должен изменять размер, независимо от того, что происходит с координатором. Если коэффициент больше нуля, он интерпретируется как доля общего размера относительно долей других вложенных в координатор элементов (это аналогично тому, как управляет шириной текстовых полей `wx.StatusBar`). Если все вложенные элементы координатора имеют одинаковый коэффициент, все они изменяют размеры пропорционально общей части пространства, которое остаётся после размещения элементов фиксированного размера. В нашем случае, 0 для панели управления указывает, что если пользователь растягивает фрейм, то панель не должна изменять горизонтальный размер, а 1 для чертежного окна означает, что все изменения размера относятся к нему.

Третий аргумент в методе `Add()` – это еще один флаг битовой маски (bitmask). Подробное описание возможных значений флага будет дано в этой главе ниже. Значение `wx.EXPAND` - одна из нескольких величин, которая управляет тем, как элемент изменяет размер по оси перпендикулярной основной оси линейного координатора; в данном случае, что происходит, когда фрейм изменяет вертикальный размер. Использование флага `wx.EXPAND` указывает координатору изменять размеры вложенного элемента, таким образом, чтобы полностью заполнять доступное пространство. Другие возможные значения позволяют изменять размеры вложенного элемента пропорционально или с выравниванием на определенную часть координатора. Рисунок 6.7 разъясняет, каким параметром управляется каждое направление изменения размера.

Результат этих установок такой, что когда Вы запускаете фрейм с таким линейным координатором, любое изменение размера в горизонтальном направлении изменяет размер окна схемы, а панель управления остаётся неизменной. Изменение размера по вертикали вызывает расширение или сжатие по вертикали обоих встроенных окон.

Указанный в листинге 6.8 класс `ControlPanel` использует комбинацию сеточного и линейного координаторов. Листинг 6.9 содержит код этого класса.

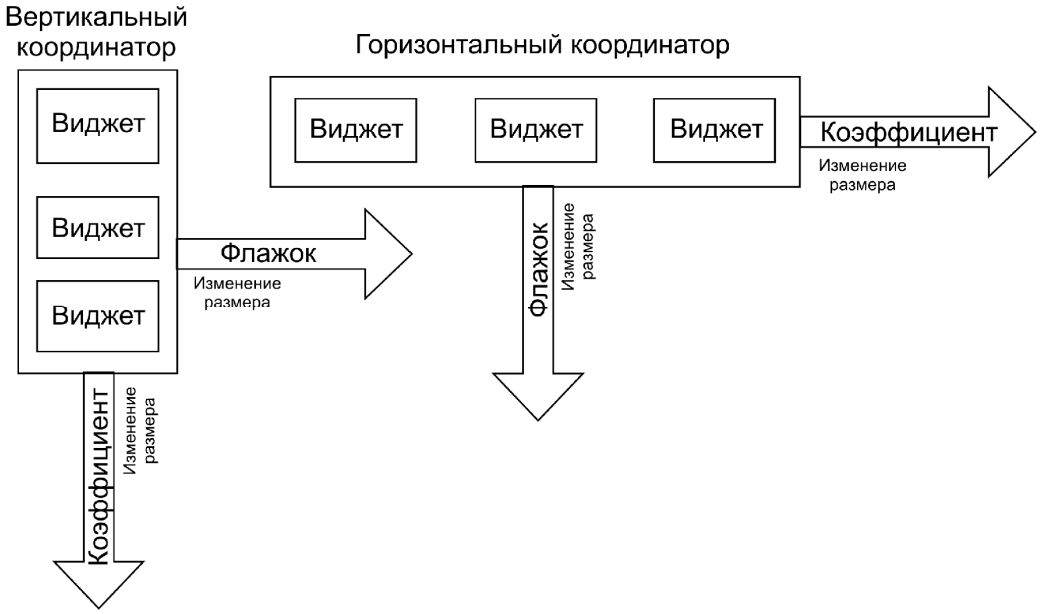

Рисунок 6.7 Аргументы, определяющие режим изменения размеров в каждом направлении

Листинг 6.9 Класс панели управления с сеточным и линейным координаторами

```
class ControlPanel(wx.Panel):

    BMP_SIZE = 16
    BMP_BORDER = 3
    NUM_COLS = 4
    SPACING = 4

    colorList = ('Black', 'Yellow', 'Red', 'Green', 'Blue', 'Purple',
                 'Brown', 'Aquamarine', 'Forest Green', 'Light Blue',
                 'Goldenrod', 'Cyan', 'Orange', 'Navy', 'Dark Grey',
                 'Light Grey')

    maxThickness = 16

    def __init__(self, parent, ID, sketch):
        wx.Panel.__init__(self, parent, ID,
                          style=wx.RAISED_BORDER)
        self.sketch = sketch
        buttonSize = (self.BMP_SIZE + 2 * self.BMP_BORDER,
                      self.BMP_SIZE + 2 * self.BMP_BORDER)
        colorGrid = self.createColorGrid(parent, buttonSize)
        thicknessGrid = self.createThicknessGrid(buttonSize)
        self.layout(colorGrid, thicknessGrid)

    def createColorGrid(self, parent, buttonSize):
        self.colorMap = {}
        self.colorButtons = {}
        colorGrid = wx.GridSizer(cols=self.NUM_COLS, hgap=2,
```

1 Создание сетки цветов

```
            vgap=2)
        for eachColor in self.colorList:
            bmp = parent.MakeBitmap(eachColor)
            b = buttons.GenBitmapToggleButton(self, -1, bmp,
                  size=buttonSize)
            b.SetBezelWidth(1)
            b.SetUseFocusIndicator(False)
            self.Bind(wx.EVT_BUTTON, self.OnSetColour, b)
            colorGrid.Add(b, 0)
            self.colorMap[b.GetId()] = eachColor
            self.colorButtons[eachColor] = b
        self.colorButtons[self.colorList[0]].SetToggle(True)
        return colorGrid

    def createThicknessGrid(self, buttonSize):
        self.thicknessIdMap = {}
        self.thicknessButtons = {}
        thicknessGrid = wx.GridSizer(cols=self.NUM_COLS, hgap=2,
            vgap=2)
        for x in range(1, self.maxThickness + 1):
            b = buttons.GenToggleButton(self, -1, str(x),
                  size=buttonSize)
            b.SetBezelWidth(1)
            b.SetUseFocusIndicator(False)
            self.Bind(wx.EVT_BUTTON, self.OnSetThickness, b)
            thicknessGrid.Add(b, 0)
            self.thicknessIdMap[b.GetId()] = x
            self.thicknessButtons[x] = b
        self.thicknessButtons[1].SetToggle(True)
        return thicknessGrid

    def layout(self, colorGrid, thicknessGrid):
        box = wx.BoxSizer(wx.VERTICAL)
        box.Add(colorGrid, 0, wx.ALL, self.SPACING)
        box.Add(thicknessGrid, 0, wx.ALL, self.SPACING)
        self.SetSizer(box)
        box.Fit(self)

    def OnSetColour(self, event):
        color = self.colorMap[event.GetId()]
        if color != self.sketch.color:
            self.colorButtons[self.sketch.color].SetToggle(False)
        self.sketch.SetColor(color)

    def OnSetThickness(self, event):
        thickness = self.thicknessIdMap[event.GetId()]
        if thickness != self.sketch.thickness:
            self.thicknessButtons[self.sketch.thickness].SetToggle
                (False)
        self.sketch.SetThickness(thickness)
```

2 **Создание сетки размеров толщины**

3 **Группировка сеток**

Примечание переводчика: *1) Не забудьте включить в модуль, в котором определяется класс* `ControlPanel`, *следующий оператор импорта:*

```
import wx.lib.buttons as buttons
```

Он делает доступными классы общих кнопок, обсуждаемые в главе 7.
2) Класс `ControlPanel` *может быть включён в основной модуль с классом* `SketchFrame`*, а также в отдельный модуль, например с именем* `ControlPanel.py`*.*
Во втором случае в основной модуль необходимо добавить оператор импорта:

```
from ControlPanel import ControlPanel
```

1. Метод `createColorGrid()` строит сеточный координатор, который содержит кнопки установки цвета. Сначала мы создаем сам координатор, определяя для него четыре колонки. Как только установлен подсчет колонок, кнопки будут размещены слева направо, сверху вниз. Затем мы берем список цветов и создаем кнопку для каждого цвета. Внутри цикла for мы создаем квадратное битовое изображение соответствующего цвета и создаем кнопку-переключатель с этим битовым изображением, используя отдельный класс из определенного в библиотеке wxPython набора классов кнопочных виджетов. Затем мы подключаем к кнопке событие и добавляем ее в сетку. После это мы добавляем ее в несколько словарей, чтобы в последующем коде проще соотносить цвет, идентификатор (ID) и кнопку. Нам не нужно указывать размещение кнопки в пределах сетки – координатор делает это за нас.

2. Метод `createThicknessGrid()` почти идентичен методу для сетки цветов. Фактически, предприимчивый программист мог бы объединить их в общую служебную функцию. Создается сеточный координатор, и поочередно добавляются шестнадцать кнопок, при этом они тщательно выравниваются на экране.

3. Для размещения сеток друг над другом мы используем вертикальный линейный координатор. Второй аргумент для каждой сетки имеет значение 0, это указывает на то, что сеточные координаторы не должны изменять размер, когда панель управления растягивается по вертикали. (Раз панель управления не изменяет горизонтальный размер, нам не нужно определять ее горизонтальное поведение.) Этот пример задействует в методе `Add()` четвертый аргумент, который определяет ширину установленной вокруг элемента границы, в нашем случае указана переменная `self.SPACING`. Значение `wx.ALL` третьего аргумента – это один из набора флагов, управляющий тем, какими сторонами касаться границы. Естественно, `wx.ALL` указывает на то, что граница будет касаться всех четырех сторон объекта. Вконце мы вызываем метод линейного координатора `Fit()` с панелью управления в качестве аргумента. Этот метод заставляет панель управления изменить свои размеры, чтобы соответствовать минимально необходимому для координатора размеру. Обычно Вы будете вызывать этот метод при создании окна, в котором используются координаторы, это гарантирует, что внешнее окно будет иметь достаточные размеры для заключения координатора.

Базовый класс `wx.Sizer` содержит несколько методов общих для всех координаторов. Таблица 6.9 перечисляет наиболее употребительные из этих методов.

Таблица 6.9 Методы класса `wx.Sizer`

Функция	Описание

`Add(window, proportion=0, flag=0, border=0, userData=None)`
`Add(sizer, proportion=0, flag=0, border=0, userData=None)`
`Add(size, proportion=0, flag=0, border=0, userData=None)`
	Помещает элемент в координатор. Первая версия помещает объект wxWindow, вторая – вложенный координатор. Третья версия добавляет используемое в качестве разделителя пустое пространство, для которого действуют те же правила, что и при позиционировании окна. Аргумент `proportion` управляет величиной размера, на которую окно изменяется относительно других окон, что имеет значение только для `wx.BoxSizer`. Аргумент `flag` является битовым изображением с множеством других флагов для выравнивания, задания позиции границы и приращения. Полный список находится в главе 11. Аргумент `border` определяет в пикселях количество устанавливаемого пространства вокруг окна или координатора. `userData` позволяет Вам ассоциировать с объектом некоторые данные, например при помощи подкласса, что даёт возможность получить дополнительную информацию при изменении размеров.

`Fit(window)` `FitInside(window)`	Заставляет передаваемое в качестве аргумента окно изменить размеры до минимальных размеров координатора. Аргумент обычно является окном, использующим координатор. Метод `FitInside()` аналогичен, но вместо изменения экранного вида окна, изменяется только его внутреннее представление. Он используется для содержимого окна панели прокрутки, чтобы выполнить показ полосы прокрутки.
`GetSize()`	Возвращает размер координатора в виде объекта `wx.Size`.
`GetPosition()`	Возвращает позицию координатора в виде объекта `wx.Point`.
`GetMinSize()`	Возвращает минимальный размер, необходимый для корректного размещения координатора в виде объекта `wx.Size`.
`Layout()`	Заставляет координатор программно пересчитать размер и позицию своих вложенных объектов. Вызывается после динамического добавления или удаления вложенного объекта.
`Prepend(...)`	Идентичен методу `Add()` (все три версии, но новый объект помещается в начало списка координатора).

`Remove(window)`
`Remove(sizer)`
`Remove(nth)`
	Удаляет объект из координатора. В зависимости от версии, удаляется или определенный объект или объект с порядковым номером `nth` в списке координатора. Если это сделано после запуска, вызовите вслед за этим метод `Layout()`.

`SetDimension(x, y, width, height)`	Принудительно устанавливает размер координатора и предписывает всем вложенным объектам самостоятельно изменить свои размеры.

За подробной информацией об обычных и вложенных координаторах обращайтесь к главе 11.

6.4.2 *Как построить диалог About?*

Диалог About является хорошим примером экранного диалога, отображающего более сложную информацию, чем в простом окне сообщения, и не требует дополнительной функциональности. В этом случае в качестве простого способа отображения стилизованного текста Вы можете использовать класс `wx.html.HtmlWindow`. В действительности, `wx.html.HtmlWindow` мощнее, чем мы здесь демонстрируем, и включает методы для управления подробным взаимодействием пользователя и предоставления. Возможности `wx.html.HtmlWindow` раскрывает глава 16. Листинг 6.10 отображает класс, создающий диалог About, в котором используется HTML.

Листинг 6.10 Использование `wx.html.HtmlWindow` в качестве диалога About

```
class SketchAbout(wx.Dialog):
    text = '''
<html>
<body bgcolor="#ACAA60">
<center><table bgcolor="#455481" width="100%" cellspacing="0"
cellpadding="0" border="1">
<tr>
    <td align="center"><h1>Sketch!</h1></td>
</tr>
</table>
</center>
<p><b>Sketch</b> is a demonstration program for
<b>wxPython In Action</b>
Chapter 6. It is based on the SuperDoodle demo included
with wxPython, available at http://www.wxpython.org/
</p>

<p><b>SuperDoodle</b> and <b>wxPython</b> are brought to you by
<b>Robin Dunn</b> and <b>Total Control Software</b>, Copyright
&copy; 1997-2006.</p>
</body>
</html>
'''

    def __init__(self, parent):
        wx.Dialog.__init__(self, parent, -1, 'About Sketch',
                           size=(440, 400) )
        html = wx.html.HtmlWindow(self)
```

```
        html.SetPage(self.text)
        button = wx.Button(self, wx.ID_OK, "Okay")

        sizer = wx.BoxSizer(wx.VERTICAL)
        sizer.Add(html, 1, wx.EXPAND|wx.ALL, 5)
        sizer.Add(button, 0, wx.ALIGN_CENTER|wx.ALL, 5)

        self.SetSizer(sizer)
        self.Layout()
```

Большая часть этого листинга представлена строкой HTML, которая имеет определенный формат и содержит шрифтовые теги. Данный диалог является комбинацией `wx.html.HtmlWindow` и кнопки с идентификатором `wx.ID_OK`. Нажатие кнопки, как и в любом другом диалоге, автоматически закрывает окно. Для управления выравниванием использован вертикальный линейный координатор.

Рисунок 6.8 отображает полученный в результате диалог.

Рисунок 6.8
Диалог About с HTML

Для того чтобы использовать данный диалог, создайте, как показано ниже, пункт меню и его обработчик:

```
def OnAbout(self, event):
    dlg = SketchAbout(self)
    dlg.ShowModal()
    dlg.Destroy()
```

Примечание переводчика: *Для поддержки `HtmlWindow` в классе `SketchAbout` не забудьте импортировать модуль `wx.html`.*

6.4.3 Как построить splash-окно

Отображение привлекательного splash-окна придаст Вашему приложению профессиональный вид. Оно может также занять пользователя, пока Ваше

приложение выполняет трудоемкую настройку. В wxPython при помощи класса `wx.SplashScreen` довольно легко построить splash-окно на основе любого битового изображения. Splash-окно может быть выведен на заданный период времени, и вне зависимости от того, установлено время или нет, экран всегда закрывается, при щелчке на нем пользователя. Этот класс практически полностью реализован своим конструктором:

```
wx.SplashScreen(bitmap, splashStyle, milliseconds, parent, id,
    pos=wx.DefaultPosition, size=wx.DefaultSize,
    style=wx.SIMPLE_BORDER|wx.FRAME_NO_TASKBAR|wx.STAY_ON_TOP)
```

Таблица 6.10 определяет параметры конструктора `wx.SplashScreen`.

Таблица 6.10 Параметры конструктора `wx.SplashScreen`

Параметр	Описание
`bitmap`	Отображаемое на экране битовое изображение `wx.Bitmap`
`splashStyle`	Стиль битового изображения, который может быть комбинацией следующих флагов: `wx.SPLASH_CENTRE_ON_PARENT`, `wx.SPLASH_CENTRE_ON_SCREEN`, `wx.SPLASH_NO_CENTRE`, `wx.SPLASH_TIMEOUT`, `wx.SPLASH_NO_TIMEOUT`. Все названия достаточно наглядны.
`milliseconds`	Если в `splashStyle` указан флаг `wx.SPLASH_TIMEOUT`, этот параметр определяет время задержки в миллисекундах, в течении которого splash-окно будет отображаться.
`parent`	Родительское окно. Как правило имеет значение `None`.
`id`	Идентификатор окна, обычно имеет значение -1.
`pos`	Определяет позицию splash-окна на экране, если в `splashStyle` указан флаг `wx.SPLASH_NO_CENTRE`.
`size`	Размер splash-окна. Обычно Вам не нужно его указывать, так как используется размер битового изображения.
`style`	Обычный стиль фрейма wxPython, его значение по умолчанию, как правило, содержит всё, что Вам необходимо.

Листинг 6.11 показывает код для splash-окна. В данном случае, мы заменили класс `wx.PySimpleApp` прикладным подклассом `wx.App`.

Листинг 6.11 Код splash-окна

```
class SketchApp(wx.App):

    def OnInit(self):
        image = wx.Image("splash.bmp", wx.BITMAP_TYPE_BMP)
        bmp = image.ConvertToBitmap()
        wx.SplashScreen(bmp, wx.SPLASH_CENTRE_ON_SCREEN |
                wx.SPLASH_TIMEOUT, 1000, None, -1)
```

```
        wx.Yield()

        frame = SketchFrame(None)
        frame.Show(True)
        self.SetTopWindow(frame)
        return True
```

Обычно splash-окно объявляется в методе `OnInit` при запуске приложения. Splash-окно отображается на экране до тех пор, пока на нем не будет сделан щелчок, или пока не закончится время задержки. В нашем случае, splash-окно отображается в центре экрана, в течение одной секунды. Вызов `Yield()` важен, поскольку он дает возможность всем ожидающим обработки событиям обработаться до того, как будет продолжена дальнейшая работа. В данном случае, он гарантирует, что прежде, чем приложение продолжит запуск, splash-окно получит и обработает свое начальное событие прорисовки.

Примечание переводчика: *В рассматриваемом примере при добавлении splash-окна целесообразно модифицировать инициилизатор приложения к виду:*

```
if __name__ == '__main__':
    app = SketchApp()
    app.MainLoop()
```

6.5 Резюме

Большинство программ wxPython использует общие элементы, такие как, меню, панели инструментов и splash-окна. Их использование делает Вашу программу более практичной и придает ей профессиональный вид. В этой главе мы использовали простое приложение для зарисовок и расширили его панелью инструментов, панелью состояния, строкой меню, общими диалогами, сложной компоновкой, окном About и splash-окном.

- Используя контекст устройства, Вы можете рисовать непосредственно в окне wxPython. Различные типы окон требуют различных классов контекстов устройства, однако их всех объединяет общий API. Контексты устройств для улучшения плавности отображения могут быть буферизованы.
- Внизу фрейма может автоматически создаваться панель состояния. Она может содержать одно или более текстовых полей, размер которых изменяется согласованно или устанавливается независимо друг от друга.
- Меню могут содержать вложенные подменю, а пункты меню могут иметь состояния переключателей. Панели инструментов воспроизводят те же типы событий, что и строки меню и предназначаются для облегчения группирования инструментальных кнопок.

- Открытие и сохранение Ваших данных может управляться стандартным диалогом `wx.FileDialog`. Цвета могут быть выбраны при помощи диалога `wx.ColourDialog`.
- Сложные виды компоновки элементов интерфейса выполняются при помощи координаторов без явного размещения каждого виджета. Координатор автоматически размещает свои дочерние объекты в соответствии с принятыми правилами. Координаторы содержат в своем составе `wx.GridSizer`, размещающий объекты на двумерной сетке и `wx.BoxSizer`, выравнивающий элементы в одну линию. Координаторы могут быть вложенными, а также могут управлять поведением своих дочерних элементов при растяжении.
- При помощи класса `wx.html.HtmlWindow` могут быть созданы простейшие диалоги и в том числе диалог About. Splash-окна создаются с помощью класса `wx.SplashScreen`.

В Части 1 мы охватили основные понятия wxPython и некоторые обобщенные задачи. В Части 2 мы используем все тот же формат «вопрос-ответ», но уже будем ставить углубленные вопросы о характере и функциональности инструментального комплекта wxPython.