33

Вывод графики с помощью GDI+

Это третья из восьми глав, посвященных взаимодействию пользователя с .NET Framework. Глава 31 была сосредоточена на таких темах, как отображение диалоговых окон, окон SDI и MDI, а также размещение различных элементов управления, в числе которых кнопки, текстовые поля и окна списков. В главе 32 мы рассматривали работу с данными в Windows Forms с применением различных элементов управления для отображения данных из самых разнообразных источников, которые могут встретиться в процессе разработки приложений.

Хотя все эти стандартные элементы управления достаточно мощны и адекватны для построения полноценного пользовательского интерфейса большинства приложений, все же бывают ситуации, когда может понадобиться дополнительная гибкость. Например, вам может потребоваться нарисовать текст в определенном шрифте в определенном, точно указанном месте окна, отобразить изображение без применения элемента управления `PictureBox` либо программно воспроизвести какие-то рисунки или другую графику. Стандартные элементы управления, описанные в главе 31, ничего этого делать не позволяют. Чтобы отобразить такого рода вывод, приложение должно иметь возможность проинструктировать операционную систему о том, что должно быть отображено и в каком месте его окна.

А потому в данной главе мы изучим, как можно рисовать разнообразные элементы, включая:

- принципы рисования;
- линии и простые контуры;
- изображения BMP и другие графические файлы;
- текст;
- вопросы печати.

В процессе нам придется использовать множество вспомогательных объектов, включая перья (pens) для определения характеристик линий, кисти (brushes) для определения закраски областей и шрифты (fonts) для определения вида символов текста. Мы также углубимся в некоторые детали относительно того, как устройства интерпретируют и отображают различные цвета.

Начнем мы, однако, с обсуждения технологии под названием *GDI+*. GDI+ состоит из набора базовых классов .NET, которые могут управлять произвольным рисованием на экране. Эти классы предназначены для отправки соответствующих управляющих инструкций драйверам графических устройств с тем, чтобы обеспечивать размещение корректного вывода на экран (или печать жесткой копии).

Основные принципы рисования

Этот раздел посвящен описанию основных принципов, которые необходимо понимать для того, чтобы приступить к рисованию на экране. Начнем его с общего обзора GDI — технологии, лежащей в основе GDI+, а также покажем, как связаны между собой GDI и GDI+. Затем рассмотрим несколько примеров.

GDI и GDI+

Вообще говоря, одной из сильных сторон Windows — да и всех современных операционных систем — является их способность абстрагироваться от деталей, характеризующих конкретные устройства, без указаний разработчика. Например, вам не нужно понимать работу драйвера устройства жесткого диска для того, чтобы программно читать или записывать файлы на диске. Вы просто вызываете соответствующие методы в соответствующих классах .NET (или во времена, предшествующие появлению .NET, — функции Windows API). Этот принцип также справедлив в отношении рисования. Когда компьютер рисует нечто на экране, он делает это путем отправки команд видеокарте. Однако на рынке присутствуют многие сотни разнообразных видеокарт, большинство из которых имеют отличный от других набор команд и возможностей. Если бы вам пришлось принимать это во внимание и писать специфический код для каждого видеодрайвера, то написание обычного приложения стало бы практически невозможной задачей. Вот почему, начиная с ранних версий Windows, появился интерфейс графических устройств (graphical device interface — GDI).

GDI обеспечивает уровень абстракции, скрывая разницу между различными видеокартами. Вы просто вызываете функцию Windows API, чтобы выполнить специфическую задачу, а GDI внутри себя самостоятельно решает, как заставить определенную клиентскую видеокарту выполнить то, что необходимо при запуске определенного фрагмента кода. Мало того, если у клиента есть несколько устройств отображения, например, мониторов и принтеров, GDI обеспечивает практически одинаковый результат при выводе одного и того же изображения на экран и принтер. Если клиент желает напечатать нечто вместо отображения его на экране, ваше приложение просто должно сообщить системе, что устройством вывода будет принтер, после чего вызывать те же функции API точно таким же образом.

Как видим, объект контекста устройства (device context — DC; скоро мы рассмотрим его) — это очень мощный объект, и возможно, вас удивит, что под GDI *все* операции рисования должны выполняться через контекст устройства. DC используется даже в операциях, которые не требуют рисования на экране или любом другом аппаратном устройстве — таких как модификация графических изображений в памяти.

Хотя GDI предлагает относительно высокоуровневый программный интерфейс для разработчиков, он все же базируется на старом Windows API, с функциями в стиле C. GDI+ — это уровень, находящийся между GDI и приложением, которые предоставляет более интуитивно понятную, основанную на наследовании объектную модель. Хотя GDI+ — это в основном оболочка вокруг GDI, тем не менее, Microsoft посредством GDI+ предлагает ряд новых возможностей и увеличенную производительность по сравнению с некоторыми старыми средствами GDI.

Часть базовой библиотеки классов .NET, связанная с GDI+, огромна, и в этой главе мы лишь вкратце сможем коснуться основных ее возможностей, поскольку любая попытка раскрыть больше, чем крошечную часть библиотеки, превратила бы эту главу в огромное руководство, перечисляющее классы и методы. Гораздо важнее понять фундаментальные принципы, касающиеся рисования, чтобы вы смогли получить возможность при необходимости легко расширить свои знания. Полные списки всех классов и методов, доступных в GDI+, конечно же, содержатся в документации SDK.

Разработчикам на Visual Basic 6, скорее всего, концепции рисования покажутся незнакомыми, потому что Visual Basic 6 сосредоточен на элементах управления, которые выполняют рисование самостоятельно. Разработчики на C++/MFC, однако, окажутся на более знакомой территории, поскольку MFC требует от разработчиков более подробного управления процессом рисования с применением GDI. Однако даже если вы имеете обширный опыт работы с классическим GDI, все равно в этой главе найдете для себя много нового.

Пространства имен GDI+

В табл. 33.1 представлен обзор основных пространств имен, в которых находятся базовые классы GDI+.

Таблица 33.1. Основные пространства имен для базовых классов GDI+

Пространство имен	Описание
`System.Drawing`	Содержит большинство классов, структур, перечислений и делегатов, обеспечивающих базовую функциональность рисования.
`System.Drawing.Drawing2D`	Представляет основную поддержку для двумерной и векторной графики, включая сглаживание, геометрические трансформации и графические пути.
`System.Drawing.Imaging`	Содержит различные классы, обеспечивающие манипуляции с графическими изображениями (битовые карты, файлы GIF и тому подобное).
`System.Drawing.Printing`	Содержит классы, имеющие отношение к печати и предварительному просмотру выводимых на печать изображений.
`System.Drawing.Design`	Включает некоторые предопределенные диалоговые окна, таблицы свойств и другие элементы интерфейса, имеющие отношение к расширению пользовательского интерфейса времени проектирования.
`System.Drawing.Text`	Включает классы для выполнения более сложных манипуляций со шрифтами и семействами шрифтов.

Отметим, что почти все классы и структуры, использованные в этой главе, относятся к пространству имен `System.Drawing`.

Контексты устройств и объект `Graphics`

В GDI способ идентификации устройства, на которое нужно направить вывод, заключается в обращении к объекту, называемому *контекстом устройства* (device context — DC). DC сохраняет информацию об определенном устройстве и может транслировать вызовы функций программного интерфейса GDI в конкретные команды, направляемые устройствам. Вы также можете опросить контекст устройства на предмет того, какие возможности он предоставляет (например, может ли принтер печатать в цвете или же только в черно-белом изображении), дабы соответствующим образом откорректировать вывод. Если вы пытаетесь заставить устройство делать что-то такое, что оно не способно сделать, то DC обычно обнаруживает это и предпринимает

соответствующие действия (которые, в зависимости от ситуации, могут означать генерацию исключения либо модификацию запроса таким образом, чтобы получить как можно более близкий результат в рамках возможностей данного устройства).

Однако DC не только имеет дело с аппаратным устройством. Он служит в качестве моста между приложением и Windows, и принимает во внимание любые требования и ограничения, налагаемые на рисование Windows. Например, если Windows знает, что необходимо перерисовать лишь часть окна вашего приложения, DC перехватит и отменит попытки рисования вне этой области. Благодаря связи DC с Windows, работа через контекст устройств может упростить ваш код и в других отношениях.

Например, аппаратным устройствам необходимо сообщать, где следует рисовать объекты, и обычно им нужны координаты, отсчитываемые относительно верхнего левого угла экрана (или другого выходного устройства). Однако приложения обычно отображают нечто в клиентской области (области, зарезервированной для рисования) собственного окна, возможно, используя собственную систему координат. Поскольку окно может быть позиционировано в любом месте экрана, и пользователь в любой момент может его переместить, преобразования между этими системами координат могут оказаться непростой задачей. Однако DC всегда знает, где находится ваше окно, и может выполнять такие преобразования автоматически.

В GDI+ контекст устройства помещен в оболочку базового класса .NET с именем `System.Drawing.Graphics`. Большая часть рисования выполняется вызовом методов экземпляра `Graphics`. Фактически, поскольку класс `Graphics` отвечает за выполнение большинства операций рисования, очень немного в GDI+ происходит такого, что не касалось бы тем или иным образом экземпляра `Graphics`, а потому понимание того, как управлять этим объектом, является ключом к пониманию того, как рисовать на устройствах отображения с помощью GDI+.

Рисование контуров

Этот раздел мы начнем с короткого примера — `DisplayAtStartup`, чтобы показать рисование в главном окне приложения. Все примеры этой главы созданы в Visual Studio 2008 как Windows-приложения на C#. Вспомним, что для проектов этого типа мастер, генерирующий код, создаст класс с именем `Form1`, унаследованный от `System.Windows.Form`, который будет представлять главное окно приложения. Также будет сгенерирован класс `Program` (находящийся в файле `Program.cs`), представляющий главную стартовую точку приложения. Если не указано иначе, во всех последующих примерах кода новый или модифицированный код будет означать код, который мы добавим к коду, сгенерированному мастером. (Коды примеров доступны на прилагаемом компакт-диске.)

> *При использовании .NET, когда мы говорим о приложениях, отображающих различные элементы управления, термин "форма" в основном заменяет "окно" и означает прямоугольный объект, занимающий место на экране по требованию приложения. В настоящей главе мы отдаем предпочтение термину "окно", потому что в контексте ручного рисования элементов оно более осмысленно. Но мы также будем говорить о формах, ссылаясь на классы .NET, которые порождают экземпляры форм/окон. И, наконец, мы будем использовать термин "рисование", описывая процесс отображения некоторых элементов на экране или другом выходном устройстве.*

В первом примере будет просто создана форма и выполнено рисование на ней в теле конструктора. Отметим, что этот способ не является ни лучшим, ни, к тому же, правильным способом рисования на экране — вы быстро обнаружите, что с этим примером связана проблема, состоящая в том, что он не в состоянии перерисовать что

бы то ни было после запуска. Однако этот пример позволяет проиллюстрировать несколько важных моментов, касающихся рисования, не обременяя слишком большим объемом работы.

Для построения этого примера создадим в Visual Studio 2008 приложение Windows. Сначала установим белый фон формы. В данном примере эта строка будет находиться после метода `InitializeComponent()`, так что Visual Studio 2008 сможет распознать ее и изменить представление формы в конструкторе. Чтобы найти метод `InitializeComponent()`, нужно щелкнуть на кнопке Show All Files (Показать все файлы) в проводнике решений (Solution Explorer) Visual Studio, затем щелкнуть на значке + рядом с файлом `Form1.cs`. Здесь мы найдем файл `Form1.Designer.cs`. Именно в этом методе находится метод `InitializeComponent()`. Можно использовать представление конструктора, чтобы установить цвет фона, но это даст тот же результат за счет автоматического добавления той же самой строки:

```
private void InitializeComponent()
{
    this.components = new System.ComponentModel.Container();
    this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
    this.Text = "Form1";
    this.BackColor = System.Drawing.Color.White;
 }
```

Затем мы добавим код к конструктору `Form1`. Создадим объект `Graphics`, вызвав метод формы `CreateGraphics()`. Объект `Graphics` содержит в себе Windows DC (контекст устройства), который нам понадобится для рисования. Контекст устройства ассоциирован с дисплеем, а также с данным окном:

```
public Form1()
{
  InitializeComponent();

  Graphics dc = CreateGraphics();
  Show();
  Pen bluePen = new Pen(Color.Blue, 3);
  dc.DrawRectangle(bluePen, 0,0,50,50);
  Pen redPen = new Pen(Color.Red, 2);
  dc.DrawEllipse(redPen, 0, 50, 80, 60);
}
```

Как видите, здесь вызывается метод `Show()` для отображения окна. Это приводит к немедленному показу окна на экране, потому что невозможно выполнить никакого рисования до тех пор, пока окно не отображено. Если окно не отображено, рисовать не на чем.

И, наконец, мы выведем прямоугольник, начиная с координат (0,0) шириной и высотой 50, а также эллипс в координатах (0,50) шириной 80 и высотой 50. Отметим, что координаты (x,y) транслируются в x пикселей вправо и y пикселей вниз, начиная от левого верхнего угла клиентской области окна.

Перегрузки методов `DrawRectangle()` и `DrawEllipse()`, которые мы используем, принимают по пять параметров каждая. Первый параметр каждой из них представляет собой экземпляр класса `System.Drawing.Pen`. Объект `Pen` (перо) — один из множества поддерживающих рисование объектов — содержит информацию о том, как рисовать линии. Наше первое перо указывает, что линия должна быть синего цвета и иметь ширину 3 пикселя; второе перо сообщает, что линия должна быть красного цвета и шириной 2 пикселя. Остальные четыре параметра — это координаты и размер. Для прямоугольника они представляют координаты (x,y) левого верхнего угла,

дополненные его шириной и высотой. Для эллипса эти числа представляют то же самое, если вообразить гипотетический прямоугольник, в который вписан эллипс, а не сам эллипс. На рис. 33.1 показан результат работы этого кода. Конечно, поскольку наша книга не цветная, вы не увидите здесь разных цветов.

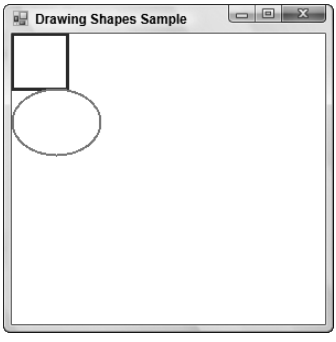

Рис. 33.1. Результат рисования прямоугольника и эллипса

Этот рисунок демонстрирует два важных момента. Во-первых, здесь ясно можно видеть расположение клиентской области окна. Она представлена белым полем, установленным через свойство `BackColor`. Прямоугольник приютился в левом верхнем углу этой области, как и можно было ожидать по его координатам (0,0). Во-вторых, заметно, что верхняя часть эллипса слегка перекрывает прямоугольник, чего нельзя было ожидать на основе указанных для него координат. В этом виновата сама Windows, поскольку она решает, где проводить линии, ограничивающие прямоугольник и эллипс. По умолчанию Windows пытается центрировать линии по границе контура, что не всегда можно сделать в точности, потому что линия должна быть нарисована по пикселям (что очевидно). Обычно граница контура проходит между пикселями. В результате линия толщиной в 1 пиксель будет нарисована *внутри* начала контура — ниже его верхней и правее левой границ, но *вне* нижней и правой сторон, а потому контуры, которые должны примыкать друг к другу, перекроются на один пиксель. Мы указали более толстые линии, а потому перекрытие больше.

Можно изменить поведение по умолчанию, установив свойство `Pen.Alignment`, как описано в документации по SDK, но пока для наших целей вполне подойдет поведение по умолчанию.

К сожалению, запустив этот пример, мы увидим, что наша форма ведет себя несколько странно. Все в порядке, если после ее появления оставить ее на месте. Также все будет нормально, если перетаскивать ее по экрану с помощью мыши. Если же попытаться минимизировать окно и затем восстановить его снова, то наши тщательно нарисованные фигуры исчезнут! То же произойдет, если перетащить какое-то другое окно над нашим примером так, чтобы оно частично перекрывало нарисованные контуры. Если это окно снова убрать, мы увидим, что исчезнет часть нашего рисунка, и мы увидим только половинку прямоугольника и половинку эллипса!

Что происходит? Проблема возникает, когда скрывается часть окна, так как Windows обычно немедленно отбрасывает всю информацию о невидимом изображении. Это приходится делать потому, что в противном случае для сохранения экранных данных потребовался бы астрономический объем памяти. Типичный компьютер может работать с видеокартой, отображающей изображение размером 1024×768 пикселей при цветовой глубине 24 бита, что подразумевает, что на хранения каждого пикселя экрана потребуется 3 байта, т.е. 2,25 Мбайт на весь экран (что означает 24-битная цветовая глубина будет пояснено далее в этой главе). Однако нередко оказывается, что пользователь работает с 10 или 20 минимизированными окнами в панели задач. В худшем случае получаем 20 окон, каждое из которых занимает полный экран в развернутом виде. Если бы Windows действительно сохраняла всю визуальную информацию этих окон, ожидая момента, когда пользователь развернет их, то понадобилось бы 45 Мбайт видеопамяти! В наше время хорошая видеокарта имеет 64, 128, 256 или 512 Мбайт памяти и справится с этим, но всего несколько лет назад 4 Мбайт видеопамяти были обычным делом, и тогда избыток пришлось бы сохранять в основной памяти компьютера. Множество людей до сих пор работают на старых машинах, неко-

торые еще с 4 Мбайт видеопамяти. Понятно, что было бы непрактично для Windows управлять пользовательским интерфейсом подобным образом.

В тот момент, когда любая часть окна скрывается, "скрытые" пиксели теряются, потому что Windows освобождает память, которую они занимали. Однако она запоминает, что часть окна была скрыта, и когда обнаруживает, что эта часть окна опять открыта, то запрашивает у приложения, владеющего окном, перерисовку его содержимого. Имеется несколько исключений из этого правила — обычно, когда речь идет о сокрытии очень небольших частей окон на короткое время (хорошим примером может служить выбор элемента главного меню, когда из него выпадает подменю, временно накрывая часть окна). Обычно же можно ожидать, что если часть окна была скрыта, то ваше приложение должно перерисовать его.

Именно это является причиной проблемы с нашим примером приложения. Мы поместили код рисования в конструктор `Form1`, который вызывается только один раз при запуске приложения, и мы не можем вызвать этот конструктор снова, чтобы перерисовать наши фигуры, когда это понадобится позднее.

При работе с элементами управления Windows Forms нет необходимости знать что-либо о том, как ими решается эта задача. Дело в том, что стандартные элементы управления достаточно сложны, и они знают, как им следует перерисовывать себя в любой момент, когда Windows попросит их об этом. Это объясняет, почему программируя элементы управления, нам вообще не приходится беспокоиться о реальном процессе из рисования. Если же мы берем на себя ответственность за отображение экрана своего приложения, то надо также позаботиться о том, чтобы приложение корректно отвечало на запросы Windows, касающиеся перерисовки всех частей его окна. В следующем разделе мы внесем необходимые изменения, чтобы сделать это.

Рисование контуров с использованием `OnPaint()`

Не беспокойтесь, если предыдущие объяснения создали у вас впечатление, что рисование собственного пользовательского интерфейса выглядит ужасающе сложной задачей. Заставить приложение перерисовать себя, когда это необходимо, на самом деле достаточно просто.

Windows уведомляет приложение о том, что требуется некоторая перерисовка, возбуждая событие `Paint`. Интересно, что класс `Form` уже имеет реализованный обработчик этого события, поэтому нам не придется добавлять его самостоятельно. Обработчик `Form1` события `Paint` содержится в виртуальном методе `OnPaint()`, которому передается единственный параметр `PaintEventArgs`. Это значит, что все, что нам необходимо сделать — это переопределить `OnPaint()`, чтобы он выполнял требуемое нам рисование.

Хотя в данном примере мы переопределим `OnPaint()`, но с тем же успехом можно было бы достичь того же результата, просто добавив еще один собственный обработчик события `Paint` (скажем, метод `Form1_Paint()`) — точно так же, как это делается с любым событием Windows Forms. Этот второй подход, возможно, более удобен, поскольку новый обработчик события можно добавить в окне свойств Visual Studio 2008, что избавляет от необходимости вводить код вручную. Однако вариант с переопределением `OnPaint()` обеспечивает дополнительную гибкость, позволяя управлять тем, когда произойдет обработка события базовым классом окна, к тому же такой подход рекомендован в документации.

В данном разделе создадим новое Windows-приложение под названием `DrwShapes`. Как и ранее, установим белый цвет фона, используя окно свойств. Кроме того, изменим текст окна на `DrawShapes Sample`. После этого добавим следующий фрагмент к сгенерированному коду класса `Form1`.

```
protected override void OnPaint( PaintEventArgs e )
{
  base.OnPaint(e);
  Graphics dc = e.Graphics;
  Pen bluePen = new Pen(Color.Blue, 3);
  dc.DrawRectangle(bluePen, 0,0,50,50);
  Pen redPen = new Pen(Color.Red, 2);
  dc.DrawEllipse(redPen, 0, 50, 80, 60);
}
```

Обратите внимание, что метод `OnPaint()` объявлен как `protected`, поскольку обычно он используется внутри класса, поэтому нет причин любому коду вне класса знать о его существовании.

`PaintEventArgs` — это наследник класса `EventArgs`, обычно используемого для передачи информации о событиях. `PaintEventArgs` имеет два дополнительных свойства, более важное из которых — это экземпляр `Graphics`, уже подготовленный и оптимизированный для рисования нужной области окна. Это значит, что нам не придется в методе `OnPaint()` вызывать `CreateGraphics()`, чтобы получить DC — нам его уже передали. Второе дополнительное свойство мы вскоре также рассмотрим; оно содержит более детализированную информацию о том, какая область окна действительно нуждается в перерисовке.

В нашей реализации `OnPaint()` первым делом мы получаем ссылку на объект `Graphics` из `PaintEventArgs`, затем рисуем наши фигуры точно так, как делали это раньше. И в конце вызываем метод `OnPaint()` базового класса. Этот шаг важен. Мы переопределили `OnPaint()` для выполнения нашего собственного рисования, но, возможно, что Windows может иметь некоторую дополнительную работу, которую необходимо выполнить в процессе рисования — вся эта работа будет выполнена в методе `OnPaint()` одного из базовых классов .NET.

В этом примере, если убрать вызов `OnPaint()` базового класса, то мы не увидим никакого особого эффекта. Однако не следует поддаваться соблазну отказаться от этого вызова. Это может нарушить нормальную работу Windows, и результаты могут быть непредсказуемыми.

Метод `OnPaint()` также вызывается при первоначальном запуске приложения, когда наше окно впервые отображается на экране. В вязи с этим нет необходимости дублировать код рисования в конструкторе.

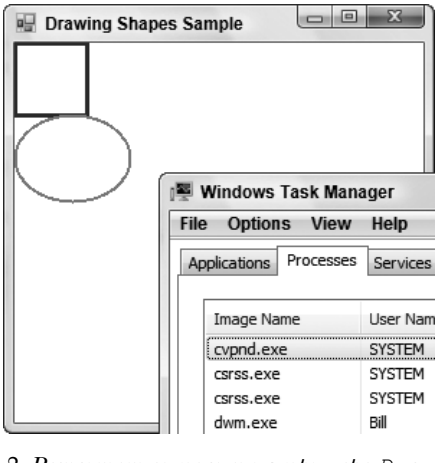

Рис. 33.2. Результат выполнения примера `DrawShapes` при перекрытии формы другим окном

Запуск этого кода изначально даст тот же результат, что и предыдущий пример, за исключением того, что теперь наше приложение ведет себя правильно при минимизации окна или сокрытии его части.

Использование области отсечения

Пример приложения `DrawShapes` из предыдущего раздела иллюстрирует главные принципы, касающиеся рисования в окнах, хотя он и не слишком эффективен. Причина связана с попыткой рисовать все в окне, независимо от того, насколько много в действительности элементов должно быть перерисовано. На рис. 33.2 показан результат выполнения примера `DrawShapes` с последующим открытием другого окна и перемещением его над формой `DrawShapes` так, что часть его перекрывается.

Пока все хорошо. Однако когда мы сдвинем перекрывшее окно так, что окно `DrawShapes` снова будет полностью видимо, Windows, как обычно, пошлет нашей форме событие `Paint`, запрашивая ее перерисовку. Прямоугольник и эллипс находятся в левом верхнем углу клиентской области, а потому оставались видимыми все время. Таким образом, на самом деле не было необходимости их все время перерисовывать вдобавок к перерисовке белого фона окна. Однако Windows не знает об этом, а потому думает, что должна сгенерировать событие `Paint`, что приводит к вызову нашей реализации `OnPaint()`. И эта реализация выполняет совершенно излишнюю перерисовку прямоугольника и эллипса.

На самом деле в этом случае фигуры не будет перерисованы благодаря контексту устройства. Windows предварительно инициализирует контекст устройства информацией об области, которую в действительности необходимо перерисовать. Во времена GDI область, помеченная для перерисовки, называлась *недействительной областью* (invalidated region), но в GDI+ эта терминология в значительной мере изменена и теперь эта область называется *областью отсечения* (clipping region). Контекст устройства распознает эту область. Таким образом, он пресекает любые попытки рисовать вне его, и не передает соответствующие команды рисования видеокарте. Звучит хорошо, но все же здесь остается потенциальная опасность снижения производительности. Невозможно предсказать, сколько обработки должен выполнить контекст устройства, прежде чем он обнаружит то, что рисование происходит за пределами недействительной области. В некоторых случаях этот объем может оказаться довольно большим, поскольку вычисление того, какие пиксели должны быть изменены и в какой цвет, может быть достаточно ресурсоемким (хотя в хороших видеокартах для этого предусмотрено аппаратное ускорение).

В результате всего этого запрос к экземпляру `Graphics` выполнить какое-то рисование вне недействительной области почти наверняка приведет к излишней трате процессорного времени и замедлит приложение. В хорошо спроектированном приложении код должен помогать контексту устройства, выполняя несколько простых проверок, чтобы убедиться, что не будет предпринято излишних попыток вызова методов экземпляра `Graphics`. В этом разделе мы закодируем новый пример `DrawShapesWithClipping`, модифицировав предыдущий `DisplayShapes`. В коде `OnPaint()` будут выполнены простые проверки, чтобы убедиться, что недействительная область пересекается с областью, в которой нужно рисовать, и только в случае, если это так, рисование будет выполнено.

Во-первых, нам нужно получить информацию об области отсечения. Для этого нам пригодится новое свойство `ClipRectangle` класса `PaintEventArgs`. Свойство `ClipRectangle` содержит координаты перерисовываемой области, оформленные в виде экземпляра структуры `System.Drawing.Rectangle`. Это достаточно простая структура — она содержит четыре свойства, которые нас интересуют: `Top`, `Bottom`,

Left и Right. Они описывают, соответственно, вертикальные координаты верхней и нижней границы прямоугольной области, а также горизонтальные координаты левой и правой ее граней.

Далее нам нужно решить, как проверять необходимость рисования. Это несложно. Вспомним, что у нас прямоугольник и эллипс целиком помещаются внутри прямоугольника, который распространяется от точки (0,0) до точки (80,130) клиентской области. В действительности даже до точки (82,132), так как мы знаем, что линии могут выйти на пиксель или два за границу. Поэтому нам нужно проверить, входит ли левый верхний угол области отсечения в пределы этого прямоугольника. Если это так, потребуется выполнить перерисовку. Если нет — можно не беспокоиться.

Код, делающий это, будет выглядеть следующим образом:

```
protected override void OnPaint( PaintEventArgs e )
{
  base.OnPaint(e);
  Graphics dc = e.Graphics;
  if (e.ClipRectangle.Top < 132 && e.ClipRectangle.Left < 82)
  {
     Pen bluePen = new Pen(Color.Blue, 3);
     dc.DrawRectangle(bluePen, 0,0,50,50);
     Pen redPen = new Pen(Color.Red, 2);
     dc.DrawEllipse(redPen, 0, 50, 80, 60);
  }
}
```

Обратите внимание — то, что отображается, ничуть не изменилось. Однако производительность увеличена за счет раннего обнаружения тех случаев, когда ничего перерисовывать не нужно. Отметим также, что в данном примере используется довольно грубая проверка того, что должно быть перерисовано. Более тонкий подход должен был бы отдельно проверять необходимость перерисовки прямоугольника и эллипса. Однако пока этого достаточно. Можно усложнить проверки в коде OnPaint(), что увеличит производительность, но при этом собственно код OnPaint() усложнится. Почти всегда имеет смысл вставить какую-то проверку, потому что вы, как автор кода, понимаете гораздо больше в том, что нужно перерисовать, чем это понимает экземпляр Graphics, который слепо исполняет команды рисования.

Измерение координат и областей

В предыдущем примере мы столкнулись с базовой структурой Rectangle, используемой для представления координат прямоугольников. GDI+ в действительности применяет несколько подобных структур для представления координат или областей.

В табл. 33.2 перечислены структуры, определенные в пространстве имен System.Drawing.

Таблица 33.2. Структуры, определенные в пространстве имен System.Drawing

Структура	Основные общедоступные свойства
Point	X, Y
PointF	X, Y
Size	Width, Height
SizeF	Width, Height
Rectangle	Left, Right, Top, Bottom, Width, Height, X, Y, Location, Size
RectangleF	Left, Right, Top, Bottom, Width, Height, X, Y, Location, Size

Обратите внимание, что многие из этих объектов включают множество других свойств, методов и перегруженных операций, которые здесь не приведены. В настоящем разделе мы поговорим только о наиболее важных из них.

Point и PointF

Point — концептуально самая простая структура. Математически она эквивалентна двумерному вектору. Она содержит два общедоступных свойства, представляющих перемещение в горизонтальном и вертикальном направлениях от некоторой начальной позиции (возможно, на экране), как показано на рис. 33.3.

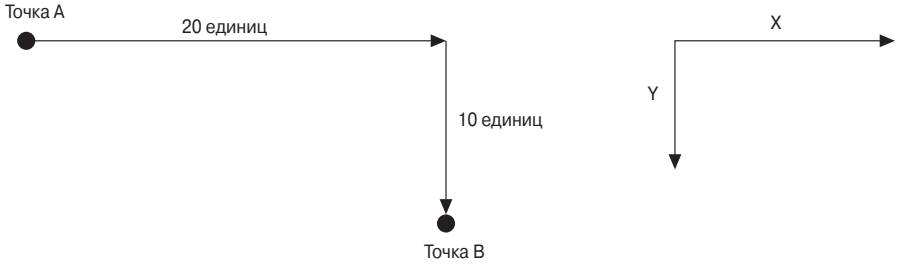

Рис. 33.3. Структура `Point`

Чтобы попасть из точки A в точку B, мы перемещаемся на 20 единиц вправо и на 10 вниз, что отмечено на диаграмме координатами x и y, поскольку так их принято называть. Следующая структура Point в программе может быть представлена так:

```
Point ab = new Point(20, 10);
Console.WriteLine("Перемещение на {0} вправо, {1} вниз", ab.X, ab.Y);
```

X и Y — свойства, доступные как для чтения, так и для записи; это значит, что можно устанавливать значения Point следующим образом:

```
Point ab = new Point();
ab.X = 20;
ab.Y = 10;
Console.WriteLine("Перемещение на {0} вправо, {1} вниз", ab.X, ab.Y);
```

Отметим, что хотя условно горизонтальная и вертикальная координаты обычно называют x и y (прописными буквами), соответствующие свойства Point называются X и Y (заглавными), поскольку в C# принято соглашение называть общедоступные свойства, начиная с заглавных букв.

Структура PointF, по сути, идентична Point, за исключением того, что X и Y имеют тип float вместо int. Структура PointF используется в тех случаях, когда координаты не обязательно являются целыми числами. Определено приведение, позволяющее неявно преобразовывать Point в PointF (напомним, что поскольку Point и PointF — структуры, это приведение включает копирование данных). Обратное преобразование не предусмотрено. Поэтому, чтобы конвертировать PointF в Point, необходимо копировать значения по отдельности либо применять один из трех методов округления: Round(), Truncate() или Celling():

```
PointF abFloat = new PointF(20.5F, 10.9F);
// преобразование к Point
Point ab = new Point();
ab.X = (int)abFloat.X;
ab.Y = (int)abFloat.Y;
Point ab1 = Point.Round(abFloat);
```

```
Point ab2 = Point.Truncate(abFloat);
Point ab3 = Point.Ceiling(abFloat);
// обратное преобразование к PointF - явное
PointF abFloat2 = ab;
```

У вас может возникнуть вопрос: в каких единицах могут выполняться измерения? По умолчанию GDI+ интерпретирует их как пиксели экрана (или принтера, если он может выводить графику); так методы объекта `Graphics` видят любые координаты, переданные им в качестве параметров. Например, точка `new Point(20,10)` описывает точку на экране, отстоящую от начала координат на 20 пикселей по горизонтали и на 10 по вертикали. Обычно пиксели измеряются от левого верхнего угла клиентской области окна, как это было во всех примерах, приведенных до сих пор. Однако так бывает не всегда. Например, может понадобиться рисовать относительно левого верхнего угла всего окна (включая рамки) или даже от левого верхнего угла всего экрана. Но в большинстве случаев, если только в документации не указано иначе, можно предположить, что отсчет пикселей идет от левого верхнего угла клиентской области окна.

Позднее вы узнаете об этом больше, когда мы будем рассматривать прокрутку, которая использует три разных системы координат — мировую (world), страничную (page) и систему координат устройства (device).

`Size` и `SizeF`

Подобно `Point` и `PointF`, размеры измеряются двумя способами. Структура `Size` применяется при использовании типов `int`, `SizeF` — при использовании `float`. В остальном `Size` и `SizeF` идентичны. В этом разделе внимание будет сосредоточено на структуре `Size`.

Во многих отношениях структура `Size` подобна `Point`. Она имеет два целочисленных свойства, представляющих расстояние по горизонтали и вертикали. Главное отличие в том, что вместо `X` и `Y` эти свойства называются `Width` и `Height`. Показанную ранее диаграмму можно представить таким кодом:

```
Size ab = new Size(20,10);
Console.WriteLine("Перемещение на {0} вдоль, {1} вниз", ab.Width, ab.Height);
```

Несмотря на то что структура `Size` математически представляет точно ту же вещь, что и `Point`, концептуально она предназначена для несколько отличающегося применения. `Point` применяется для описания местоположения чего-либо, а `Size` — для описания размера этого чего-либо. Однако поскольку `Point` и `Size` так тесно связаны, между ними поддерживается преобразование:

```
Point point = new Point(20, 10);
Size size = (Size) point;
Point anotherPoint = (Point) size;
```

В качестве примера представим прямоугольник, который мы рисовали в предыдущих примерах, с координатами левого верхнего угла (0,0) и размерами (50,50). Размер прямоугольника (50,50) может быть представлен экземпляром `Size`. Правый нижний угол также имеет координаты (50,50), но он должен быть представлен экземпляром `Point`. Чтобы увидеть разницу, предположим, что мы нарисовали прямоугольник в другом месте, так что координаты его левого верхнего угла стали (10,10):

```
dc.DrawRectangle(bluePen, 10,10,50,50);
```

Теперь координаты нижнего правого угла будут (60,60), но размер не изменится и будет по-прежнему (50,50).

Операция сложения для структур `Point` и `Size` перегружена, так что можно добавлять `Size` к `Point` и получать в результате `Point`:

```
static void Main(string[] args)
{
  Point topLeft = new Point(10,10);
  Size rectangleSize = new Size(50,50);
  Point bottomRight = topLeft + rectangleSize;
  Console.WriteLine("topLeft = " + topLeft);
  Console.WriteLine("bottomRight = " + bottomRight);
  Console.WriteLine("Size = " + rectangleSize);
}
```

Запуск этого кода, как простого консольного приложения по имени PointsAndSizes, приведет к выводу, показанному на рис. 33.4.

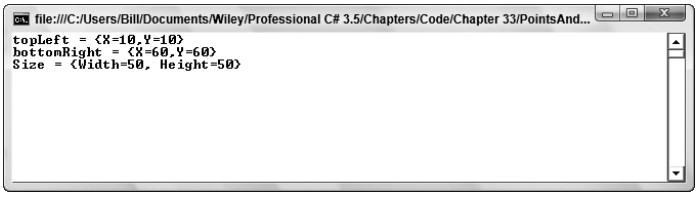

Рис. 33.4. Результат выполнения простого консольного приложения PointsAndSizes

Обратите внимание, что этот вывод также показывает, как переопределен метод ToString() для структур Point и Size с целью отображения значения в формате {X,Y}.

Можно также вычитать Size из Point, получая в результате структуру Point, а также складывать вместе две структуры Size, получая новую Size. Однако невозможно прибавить Point к другой структуре Point. Проектировщики из Microsoft решили, что сложение структур Point концептуально не имеет смысла, а потому не стали перегружать операцию +, которая позволила бы это делать.

Можно также выполнять явные приведения Point к Size и наоборот:

```
Point topLeft = new Point(10,10);
Size s1 = (Size)topLeft;
Point p1 = (Point)s1;
```

С этим приведением s1.Width присваивается значение topLeft.X, а s1.Height — значение topLeft.Y. Таким образом, s1 будет содержать (10,10), а p1 — то же самое, что и topLeft.

Rectangle и RectangleF

Эти структуры представляют прямоугольные области (обычно на экране). Так же, как Point и Size, здесь мы рассмотрим только Rectangle. Структура RectangleF в основном идентична, с тем отличием, что ее свойства имеют тип float вместо int.

Структуру Rectangle можно представить как составную, которая включает в себя структуру Point, представляющую левый верхний угол, и структуру Size, представляющую размер. Один из ее конструкторов действительно принимает в качестве параметров пару структур — Point и Size. Убедимся в этом, переписав код, рисующий прямоугольник, из примера DrawShapes:

```
Graphics dc = e.Graphics;
Pen bluePen = new Pen(Color.Blue, 3);
Point topLeft = new Point(0,0);
Size howBig = new Size(50,50);
Rectangle rectangleArea = new Rectangle(topLeft, howBig);
dc.DrawRectangle(bluePen, rectangleArea);
```

Здесь также используется альтернативная перегрузка `Graphics.DrawRectangle()`, принимающая в параметрах `Pen` и `Rectangle`.

Можно также сконструировать структуру `Rectangle`, применив координаты верхнего левого угла, ширину и высоту раздельно как отдельные числа:

```
Rectangle rectangleArea = new Rectangle(0, 0, 50, 50);
```

`Rectangle` содержит ряд свойств, доступных для чтения и записи, которые позволяют устанавливать и извлекать ее размеры в разных комбинациях. Подробности можно найти в табл. 33.3.

Таблица 33.3. Свойства структуры `Rectangle`

Свойство	Описание
`int Left`	Координата x левой грани.
`int Right`	Координата x правой грани.
`int Top`	Координата y верхней грани.
`int Bottom`	Координата y нижней грани.
`int X`	То же, что и `Left`.
`int Y`	То же, что и `Top`.
`int Width`	Ширина прямоугольника.
`int Height`	Высота прямоугольника.
`Point Location`	Левый верхний угол.
`Size Size`	Размер прямоугольника.

Следует отметить, что не все эти свойства независимы. Например, установка значения `Width` также влияет на значение `Right`.

Region

`Region` представляет область на экране, имеющую сложную форму. Например, затененная область на рис. 33.5 может быть представлена как `Region`.

Как следовало ожидать, процесс инициализации экземпляра `Region` достаточно сложен. Говоря кратко, это можно сделать либо указанием набора компонентов простой формы, образующих область, либо указав путь прохождения по ее границам. Если вам необходимо работать с подобными областями, стоит изучить класс `Region` в документации SDK.

Рис. 33.5. Пример `Region`

Замечания по поводу отладки

Теперь мы готовы к тому, чтобы разработать более сложный пример рисования. Однако сначала следует сказать пару слов об отладке. Если вы попытаетесь установить точки прерывания в коде примеров этой главы, то обнаружите, что отладка процедур рисования не так проста, как отладка прочих частей ваших программ.

Дело в том, что вход в отладчик и выход из него часто сам по себе вызывает отправку сообщения `Paint` вашему приложению. В результате установки точки прерывания внутри `OnPaint()` может получиться так, что приложение будет вновь и вновь пытаться перерисовать себя, так что станет невозможным делать что-либо еще.

Типичный сценарий выглядит следующим образом. Вы хотите понять, почему ваше приложение отображает что-то неправильно, потому помещаете точку прерывания внутрь обработчика события `OnPaint()`. Как ожидалось, в этой точке выполнение программы прерывается, и вы оказываетесь в отладчике, т.е. на передний план попадает окно из MDI-среды разработки. Очень часто разработчики настраивают эту среду на полноэкранное представление, чтобы легко видеть сразу всю отладочную информацию, а это значит, что она полностью накрывает отлаживаемое приложение.

Продвигаясь дальше, вы исследуете значения некоторых переменных, надеясь обнаружить что-то полезное. Затем нажимаете клавишу <F5>, чтобы продолжить работу приложения и увидеть, что случается, когда приложение отображает что-то еще после обработки. К сожалению, первое, что происходит — приложение перемещается на передний план, Windows обнаруживает, что форма опять стала видимой и сразу же посылает ей событие `Paint`. И, конечно же, вы снова попадаете в свою точку прерывания. Хорошо, если это то, что вам нужно. Но обычно на самом деле вы хотели бы попасть в точку прерывания позже, когда приложение нарисует что-то более интересное — возможно, после выбора какой-то команды меню для чтения файла, либо после чего-то другого, что изменит отображение. Ситуация выглядит тупиковой. Либо вам вовсе не нужно ставить точку прерывания в `OnPaint()`, либо приложение никогда не попадет в точку, находящуюся за моментом отображения начального окна запуска. Тем не менее, эту проблему можно обойти.

Если у вас большой экран, то проще всего уменьшить окно среды разработки до половины экрана вместо того, чтобы держать его полностью развернутым. Кроме того, его нужно поместить в стороне от окна вашего приложения, чтобы оно никогда не перекрывалось. К сожалению, в большинстве случаев это решение не практично, поскольку приходится делать окно среды разработки слишком маленьким (можно также подключить еще один монитор). Альтернативой, использующей тот же принцип, может быть объявление вашего приложения на время отладки “приложением верхнего уровня”. Это делается установкой свойства `TopMost` класса `Form`, что легко можно сделать в методе `InitializeComponent()`:

```
private void InitializeComponent()
{
  this.TopMost = true;
```

Это свойство также можно установить в окне свойств Visual Studio 2008.

Имея свойство `TopMost`, равное `true`, ваше приложение никогда не будет перекрыто другими окнами (за исключением другого окна с тем же свойством). Оно всегда остается поверх остальных окон, даже когда фокус получает другое приложение. Так работает системный диспетчер задач (Task Manager).

Но, даже пользуясь таким приемом, нужно быть осторожным, потому что нельзя предсказать, когда Windows по какой-то причине решит сгенерировать событие `Paint`. Если вы действительно хотите отловить некоторую проблему, которая проявляется в

методе OnPaint() в каких-то специфических случаях (например, когда приложение рисует что-то после выбора определенной команды меню и что-то при этом идет не так, как надо), тогда лучший способ сделать это — поместить внутрь OnPaint() код, который проверит некоторое условие, являющееся истинным только именно в этом специфическом случае. Затем расположите точку прерывания внутри блока if, как показано ниже:

```
protected override void OnPaint( PaintEventArgs e )
{
  // Condition() вычисляется как true, когда нужно прервать выполнение
  if ( Condition() == true)
  {
     int ii = 0; // <-- УСТАНОВИТЬ ТОЧКУ ПРЕРЫВАНИЯ ЗДЕСЬ!!!
  }
```

Это простой и быстрый способ установки условной точки прерывания.

Рисование прокручиваемых окон

Простейший пример DrawShapes работает очень хорошо, поскольку все, что необходимо нарисовать, умещается в пределах начального размера окна. В этом разделе мы рассмотрим, что нужно делать, когда это не так.

Для этого примера мы расширим пример DrawShapes так, чтобы продемонстрировать прокрутку. Чтобы все стало несколько более реалистично, начнем с создания примера под названием BigShape, в котором нарисуем прямоугольник и эллипс покрупнее. К тому же в процессе работы с этим примером мы увидим, как использовать структуры Point, Size и Rectangle для задания областей рисования. Учитывая все эти изменения, соответствующая часть класса Form1 будет выглядеть так, как показано ниже.

```
// поля-члены
Private readonly Point rectangleTopLeft = new Point(0, 0);
private readonly Size rectangleSize = new Size(200,200);
private readonly Point ellipseTopLeft = new Point(50, 200);
private readonly Size ellipseSize = new Size(200, 150);
private readonly Pen bluePen = new Pen(Color.Blue, 3);
private readonly Pen redPen = new Pen(Color.Red, 2);
protected override void OnPaint( PaintEventArgs e )
{
     base.OnPaint(e);
     Graphics dc = e.Graphics;
     if (e.ClipRectangle.Top < 350 || e.ClipRectangle.Left < 250)
     {
       Rectangle rectangleArea =
       new Rectangle (rectangleTopLeft, rectangleSize);
       Rectangle ellipseArea =
       new Rectangle (ellipseTopLeft, ellipseSize);
       dc.DrawRectangle(bluePen, rectangleArea);
       dc.DrawEllipse(redPen, ellipseArea);
     }
}
```

Отметим также, что объекты Pen, Size и Point мы объявили членами класса — это более эффективно, нежели создание новых объектов Pen всякий раз, когда нужно что-либо нарисовать, как это делалось до сих пор. Результат запуска этого примера показан на рис. 33.6.

Проблема проявляется сразу. Наши фигуры не помещаются в пределы области 300×300 пикселей.

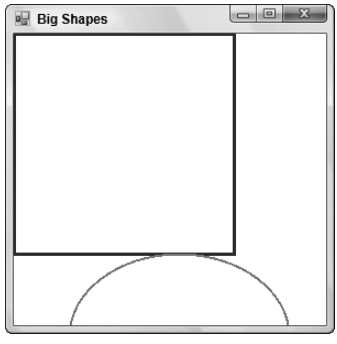

Рис. 33.6. Результат выполнения примера с крупными фигурами

Обычно, если документ слишком велик, чтобы его можно было отобразить целиком, приложения добавляют в окно линейки прокрутки, чтобы можно было прокручивать содержимое окна и просматривать его по частям. Это еще одна область применения стандартных элементов управления Windows Forms, благодаря которым мы можем возложить всю работу на базовые классы и исполняющую систему .NET. Если у вас есть форма со вставленными в нее элементами управления, то экземпляр `Form` обычно знает, где они находятся, и потому может обнаружить ситуацию, когда окно станет настолько маленьким, что понадобятся линейки прокрутки. Экземпляр `Form` добавит их для вас автоматически, и сможет корректно рисовать нужные части экрана в процессе прокрутки его содержимого. В этом случае вам ничего не придется менять в своем коде. Однако в этой главе мы берем ответственность за рисование экрана на себя, потому нам придется помочь экземпляру `Form` выводить свое содержимое при прокрутке.

Добавить линейки прокрутки очень просто. `Form` может сделать это для нас, но не знает, какого размера должно быть изображение, которое мы собираемся рисовать. (Причина их отсутствия в предыдущем примере `BigShapes` связана с тем, что Windows не знает о том, что они необходимы.) Что нужно указать — это размер прямоугольника, который охватывает область от левого верхнего угла документа (или, что равнозначно — левого верхнего угла клиентской области перед прокруткой) до такой нижней правой точки, чтобы в нее уместился весь документ (рисунок). В настоящей главе эту область будем называть областью документа. Как показано на рис. 33.7, в данном примере область документа будет иметь размер (250,350) пикселей.

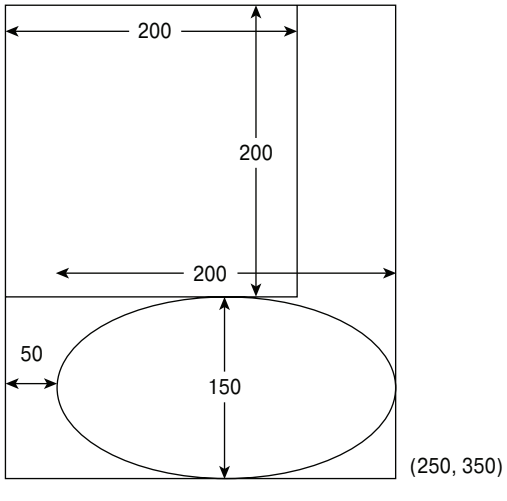

Рис. 33.7. Определение размера области документа

Сообщить форме размер документа очень легко. Для этого используется соответствующее свойство `Form.AutoScrollMinSize`. Таким образом, можно добавить следующий код либо к методу `InitializeComponent()`, либо к конструктору `Form`:

```
private void InitializeComponent()
{
   this.components = new System.ComponentModel.Container();
   this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
   this.Text = "Form1";
   this.BackColor = System.Drawing.Color.White;
   this.AutoScrollMinSize = new Size(250, 350);
}
```

Это свойство можно также установить в окне свойств Visual Studio 2008. Обратите внимание на то, что для того чтобы получить доступ к классу `Size`, нужно добавить следующий оператор `using`:

```
using System.Drawing;
```

Установить минимальный размер окна при запуске приложения и оставить его таковым — это нормально для данного конкретного примера, поскольку мы всегда знаем, каким будет размер всего экрана приложения. Наш документ никогда не изменяет своего размера в процессе работы этого приложения. Однако если принять во внимание, что приложение может, например, отображать содержимое файлов или что-то еще, для чего размер экрана может изменяться, то это свойство потребуется устанавливать в другие моменты (и в этом случае это нужно будет делать только программно, потому что окно свойств Visual Studio 2008 может помочь установить лишь начальное значение, задаваемое при конструировании формы).

Но установка свойства `AutoScrollMinSize` — только начало, и этого далеко не достаточно. На рис. 33.8 показано, как будет теперь выглядеть наше приложение — вначале мы получаем экран, корректно отображающий наши фигуры.

Отметим, что теперь форма не только корректно установила линейки прокрутки, но и правильно определила их размер, чтобы продемонстрировать, какая часть документа в данный момент отображается. Можно попытаться изменить размер окна — линейки прокрутки будут реагировать должным образом, и они даже исчезнут, если сделать окно достаточно большим, чтобы документ уместился полностью.

Однако посмотрим, что случится, если мы попытаемся прокрутить содержимое окна с помощью одной из этих линеек (рис. 33.9). Ясно, что что-то идет не так!

Причина в том, что мы не принимаем во внимание позицию линеек прокрутки в коде метода `OnPaint()`. Очень четко это можно заметить, если минимизировать и снова восстановить окно (рис. 33.10).

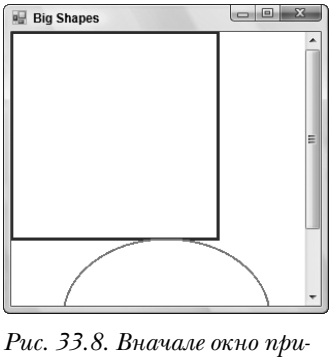

Рис. 33.8. Вначале окно приложения выглядит нормально

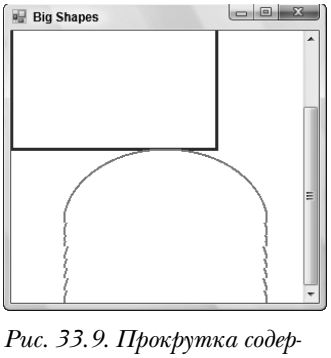

Рис. 33.9. Прокрутка содержимого окна

Фигуры рисуются, как и раньше, начиная с верхнего левого угла клиентской области — как если бы мы вообще не двигали линейку прокрутки.

Прежде, чем мы разбираться с тем, как решить эту проблему, тщательно исследуем, что мы видим на этих образах экрана.

Начнем с примера `BigShapes`, показанного на рис. 33.8. В этом примере полное окно только что перерисовано. Просмотрев код, мы видим, что он инструктирует экземпляр `Graphics` нарисовать прямоугольник, начиная с левой верхней координаты (0,0) — относительно левого верхнего угла клиентской области окна — что и происходит. Проблема состоит в том, что экземпляр `Graphics` трактует координаты как относящиеся к клиентской области окна и ничего не знает о линейках прокрутки. Наш код не пытается исправить координаты в соответствии с текущим положением линеек прокрутки. То же самое происходит с эллипсом.

Теперь посмотрим на снимок экрана, показанный на рис. 33.9. После прокрутки мы замечаем, что верхняя часть окна выглядит хорошо. Это потому, что она была нарисована при запуске приложения. Когда мы прокручиваем окно, Windows достаточно умна, чтобы обнаружить, какие из отображаемых битов экрана можно просто переместить в соответствии с положением линейки прокрутки. Это — наиболее эффективный процесс, поскольку он может использовать для этого аппаратное ускорение. Неверно отображенная часть находится в нижней трети окна. Эта часть не была нарисована при запуске приложения, поскольку тогда она выходила за пределы клиентской области. Это значит, что Windows попросит наше приложение `BigShapes` нарисовать эту область. То есть она сгенерирует событие `Paint`, передав ему координаты прямоугольника отсечения. И это приведет к вызову нашего переопределенного метода `OnPaint()`.

Одним способом решения этой проблемы может быть выражение координат для рисования относительно верхнего левого угла документа — т.е. их нужно преобразовать таким образом, чтобы выразить относительно левого верхнего угла клиентской области окна (рис. 33.11).

Чтобы сделать эту диаграмму яснее, документ на ней несколько увеличен в высоту и ширину, но это не меняет сути дела. Мы предполагаем наличие небольшой горизонтальной прокрутки наряду с вертикальной.

На рис. 33.11 тонкие прямоугольники помечают границы экранной области и границы всего документа. Толстыми линиями отображаются прямоугольник и эллипс, которые мы пытаемся нарисовать. `P` отмечает некоторую произвольную точку на рисунке, которую мы используем для примера.

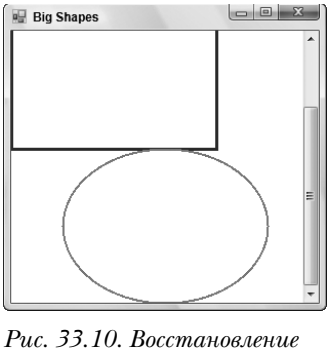

Рис. 33.10. Восстановление окна после минимизации

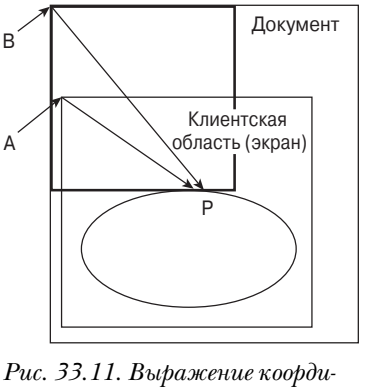

Рис. 33.11. Выражение координат для рисования относительно верхнего левого угла документа

При вызове методов рисования экземпляр Graphics применяет вектор из точки B в (скажем) точку P, выраженный как экземпляр Point. Нам нужно преобразовать его в вектор из точки A в точку P.

Проблема состоит в том, что мы не знаем, что собой представляет вектор из точки A в точку P. Мы знаем, что имеется вектор от B до P — просто координаты точки P относительно верхнего левого угла документа, т.е. позиция в документе, где нужно нарисовать точку P. Мы также знаем, что вектор от B до A — это просто величина прокрутки; это содержится в свойстве класса Form под названием AutoScrollPosition. Однако нам не известен вектор от A до P.

Если вы помните математику, то легко найдете решение этой задачи — для этого нужно просто вычесть один вектор из другого. Предположим, для примера, что чтобы попасть из B в P, нужно переместиться на 150 пикселей по горизонтали вправо и на 200 пикселей по вертикали вниз, в то время как для того, чтобы попасть из B в A, нужно переместиться на 10 пикселей вправо и на 57 вниз. Это значит, что для того, чтобы попасть из A в P, необходимо переместиться на 140 (150 – 10) вправо и на 143 (200 – 57) вниз. Чтобы еще более упростить это, класс Graphics реализует метод, который выполняет за нас эти вычисления. Он называется TranslateTransform(). Ему передаются горизонтальная и вертикальная координаты, сообщающие, где находится верхний левый угол клиентской области по отношению к верхнему левому углу документа (таким образом, свойство AutoScrollPosition представляет вектор от B к A). Устройство Graphics теперь самостоятельно обработает все координаты с учетом относительного расположения клиентской области и документа.

Для выражения этого длинного объяснения с помощью кода нам понадобится добавить всего одну строку:

```
dc.TranslateTransform(this.AutoScrollPosition.X, this.AutoScrollPosition.Y);
```

Однако в нашем примере все будет несколько сложнее, потому что также потребуется отдельно проверить, нужно ли вообще что-то рисовать, посмотрев на область отсечения. Эту проверку придется исправить с учетом текущей позиции прокрутки. Когда мы все это сделаем, наш код рисования будет выглядеть следующим образом:

```
protected override void OnPaint( PaintEventArgs e )
{
  base.OnPaint(e);
  Graphics dc = e.Graphics;
  Size scrollOffset = new Size(this.AutoScrollPosition);
  if (e.ClipRectangle.Top+scrollOffset.Width < 350 ||
       e.ClipRectangle.Left+scrollOffset.Height < 250)
  {
       Rectangle rectangleArea = new Rectangle
                    (rectangleTopLeft+scrollOffset
                    , rectangleSize);
       Rectangle ellipseArea = new Rectangle
                    (ellipseTopLeft+scrollOffset,
                    ellipseSize);
       dc.DrawRectangle(bluePen, rectangleArea);
       dc.DrawEllipse(redPen, ellipseArea);
  }
}
```

Теперь наш код работает отлично, и мы, наконец, можем получить правильный экран после прокрутки, как показано на рис. 33.12.

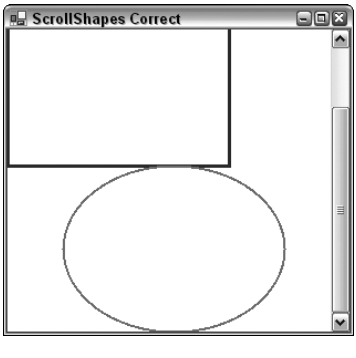

Рис. 33.12. Правильный экран после прокрутки

Мировые координаты, страничные координаты и координаты устройства

Разница между измерением положения относительно верхнего левого угла документа и относительно верхнего левого угла экрана (рабочего стола) настолько важна, что в GDI+ предусмотрены специальные наименования для этих координатных систем.

- **Мировые координаты** (world coordinates) — указывают позицию точки, измеренную в пикселях от левого верхнего угла документа.
- **Страничные координаты** (page coordinates) — указывают позицию точки, измеренную в пикселях от левого верхнего угла клиентской области.

Разработчики, знакомые с GDI, заметят, что мировые координаты соответствуют тому, что в GDI называется логическими координатами, а страничные координаты — тому, что известно как координаты устройства. Как разработчик, ранее имеющий дело с GDI, вы также отметите, что способ преобразования между логическими координатами и координатами устройства в GDI+ изменился. В GDI преобразование происходило через контекст устройства с помощью функций Windows API `LPtoDP()` *и* `DPtoLP()`*. В GDI+ имеется класс* `Control`*, от которого происходит как* `Form`*, так и все разнообразные элементы управления Windows Forms, и этот класс поддерживает всю информацию, необходимую для выполнения преобразований.*

В GDI+ также выделена третья система координат, называемая *координатами устройства* (device coordinates). Координаты устройства подобны страничным координатам, за исключением того, что в качестве единиц измерения не используются пиксели. Вместо них применяются другие единицы измерения, определяемые пользователем через вызов `Graphics.PageUnit`. Возможные единицы измерения помимо пикселей включают дюймы и миллиметры. Хотя в данной главе мы не будем использовать свойство `PageUnit`, вы можете счесть его полезным в качестве способа работы с различными разрешениями графических устройств. Например, 100 пикселей на большинстве мониторов занимают около дюйма. Однако лазерные принтеры могут иметь 1200 или более точек на дюйм (dots per inch — dpi), а это значит, что эти 100 пикселей на принтере будут выглядеть значительно меньше. Установив единицу измерения, скажем, в дюймах, и, указав размер фигуры в 1 дюйм, вы можете обеспечить одинаковый размер при ее выводе на разные устройства. Это делается так:

```
Graphics dc = this.CreateGraphics();
dc.PageUnit = GraphicsUnit.Inch;
```

К возможным единицам измерения, доступным в перечислении `GraphicsUnit`, относятся следующие:

- `Display` — определяет единицу измерения дисплея;
- `Document` — определяет единицу документа (1/300 дюйма) в качестве единицы измерения;
- `Inch` — определяет дюйм в качестве единицы измерения;
- `Millimeter` — определяет миллиметр в качестве единицы измерения;
- `Pixel` — определяет пиксель в качестве единицы измерения;
- `Point` — определяет точку принтера (1/72 дюйма) в качестве единицы измерения;
- `World` — определяет мировую систему координат в качестве единицы измерения.

Цвета

Этот раздел посвящен способам указания цвета при рисовании.

Цвета в GDI+ представлены экземплярами структур `System.Drawing.Color`. Обычно однажды создав экземпляр такой структуры, с ним не приходится ничего делать; вы просто передаете его методам, которые требуют параметра — цвета. Мы уже сталкивались с этой структурой, когда задавали цвет фона клиентской области окна в каждом их предыдущих примеров, а также когда устанавливали цвет рисования фигур. Свойство `Form.BackColor` возвращает экземпляр `Color`. В этом разделе мы рассмотрим эту структуру более детально. В частности, исследуем несколько различных способов конструирования `Color`.

Значения Red-Green-Blue (RGB)

Общее количество отображаемых на мониторе цветов поистине огромно — более 16 миллионов. Точнее сказать, их количество равно 2 в степени 24, что равно 16 777 216. Очевидно, что необходим какой-то метод индексации цветов, чтобы можно было указать, какой из них нужно использовать для отображения любого данного пикселя.

Наиболее часто применяемый способ индексации цветов определяется делением его на красный, зеленый и синий компоненты. Идея основана на теории, что любой цвет, который может различить человеческий глаз, может быть представлен определенным уровнем красного, определенным уровнем зеленого и определенным уровнем синего цветов. Эти цвета называют *компонентами*. На практике обнаружено, что если разделить яркость каждого компонента на 256 возможных уровней интенсивности, то это дает достаточно точную градацию, чтобы вывести изображение, которое человеческий глаз воспримет как имеющее фотографическое качество. Таким образом, можно специфицировать цвет, задавая уровень яркости каждого компонента в диапазоне от 0 до 255, где 0 означает отсутствие данного компонента, а 255 — его максимальную интенсивность.

Это дает нам первый способ сообщить GDI+ значение цвета. Можно указать значения красной, зеленой и синей составляющих, вызвав статическую функцию `Color.FromArgb()`. Разработчики Microsoft решили не применять конструктор для выполнения этой задачи. Причина в том, что помимо обычных компонентов RGB существуют и другие способы определения цвета. В результате в Microsoft решили, что смысл параметров, передаваемых любому конструктору, может быть открыт для неправильного понимания:

```
Color redColor = Color.FromArgb(255,0,0);
Color funnyOrangyBrownColor = Color.FromArgb(255,155,100);
Color blackColor = Color.FromArgb(0,0,0);
Color whiteColor = Color.FromArgb(255,255,255);
```

Три параметра соответствуют интенсивности красного, зеленого и синего. Эта функция имеет множество других перегрузок, часть из которых также позволяют специфицировать нечто, называемое альфа-сопряжением (alpha-blending — отсюда буква `A` в имени метода `FromArgb()`). Анализ альфа-сопряжения не входит в рамки настоящей главы, но говоря кратко, это позволяет задавать полупрозрачный цвет, комбинируя с тем, что уже имеется на экране. Это может давать некоторые симпатичные эффекты и часто используется в играх.

Именованные цвета

Конструирование `Color` с помощью `FromArgb()` — это наиболее гибкий прием, поскольку фактически дает возможность указать любой цвет, который может видеть человеческий глаз. Однако если вам нужны простые стандартные, хорошо известные цвета вроде красного или синего, то гораздо проще было бы указать имя требуемого цвета. По этой причине Microsoft также предложила большое количество статических свойств в `Color`, каждое из которых возвращает именованный цвет. Одно из них мы уже использовали, когда устанавливали белый цвет фона окна в предыдущих примерах:

```
this.BackColor = Color.White;
// дает тот же эффект, что и
// this.BackColor = Color.FromArgb(255, 255 , 255);
```

Существует несколько сотен таких цветов. Полный список приведен в документации по SDK. Они включают все простые цвета: `Red`, `White`, `Blue`, `Green`, `Black` и т.д., а также такие замечательные цвета, как `MediumAquamarine`, `LightCoral` и `DarkOrchid`. Существует также перечисление `KnownColor`, которое определяет именованные цвета.

Каждый из именованных цветов представляет точный набор значений RGB, и все они были выбраны много лет назад для применения в Internet. Идея заключалась в том, чтобы предоставить удобный набор цветов в рамках спектра, имена которых могут быть распознаны Web-браузерами, и тем самым избавить разработчиков от необходимости указывать RGB-значения в коде HTML. Еще несколько лет назад эти цвета были важны еще и потому, что ранние версии браузеров не отображали аккуратно все цвета, а потому стандартный набор определял цвета, которые должны были корректно отображаться большинством браузеров. В наши дни этот аспект менее важен, поскольку современные Web-браузеры в состоянии правильно отобразить любое RGB-значение. Также доступны безопасные к Web палитры цветов, которые обеспечивают разработчиков полными списками цветов, работающими с большинством браузеров.

Режимы отображения `Graphics` и безопасная палитра

Хотя в принципе мониторы и могут отображать любой из более чем 16 миллионов цветов RGB, на практике это зависит от того, как настроены свойства дисплея на вашем компьютере. В Windows традиционно доступны три главных цветовых режима (хотя некоторые машины могут предлагать и другие опции, в зависимости от установленного оборудования): “true color” (реалистичное цветовоспроизведение, 24 бита), “high color” (высококачественное цветовоспроизведение, 16 бит) и 256 цветов. (На некоторых графических картах в наши дни “true color” называется 32-битным режимом. Это сделано с целью оптимизации работы оборудования, хотя в этом случае для представления цветов используются лишь 24 бита из 32.)

Только режим “true color” позволяет отображать все цвета RGB одновременно. Это кажется лучшим выбором, но обходится недешево: для представления полного значения RGB требуется 3 байта — т.е. 3 байта памяти графической карты необходимы для хранения одного отображаемого пикселя. Если на первом месте стоит экономия видеопамяти (ограничение, которое теперь не так часто используется, как следовало бы), то вы можете предпочесть один из других режимов. Режим “high-color” требует 2 байта на пиксель. Этого достаточно, чтобы выделить 5 бит на каждый компонент RGB. Таким образом, вместо 256 градаций интенсивности красного получаем только 32. То же самое касается синего и зеленого, что в итоге дает нам 65 536 цветов. Этого почти достаточно, чтобы дать видимое на первый взгляд фотографическое качество, хотя полутона и будут несколько беднее.

256-цветный режим дает еще меньше цветов. Однако в этом режиме можно выбрать, какими именно должны быть эти цвета. Это обеспечивается в системе тем, что называется *палитрой*. Палитра перечисляет 256 цветов, выбранных из 16 миллионов цветов RGB. Указав однажды перечень цветов палитры, можно заставить графическое устройство отображать только эти цвета. Палитра может быть изменена в любой момент, но в каждый момент времени графическое устройство отображает только 256 цветов. Этот режим применяется в тех случаях, когда приоритетом является высокая производительность и объем используемой видеопамяти. Большинство компьютерных игр используют этот режим, и они могут обеспечить великолепно выглядящую графику за счет тщательного выбора палитры.

Вообще говоря, если устройство отображения находится в режиме "high-color" либо в 256-цветном, и запрашивается определенный цвет RGB, то оно находит ближайший математически соответствующий цвет из множества доступных к отображению. Мы говорим об этом потому, что следует иметь представление о существовании режимов цвета. Если вы рисуете нечто такое, что требует тонкого отображения полутонов, или выводите изображения с фотографическим качеством, а у пользователя не включен режим 24-битного цвета, он может не увидеть образ так, как вы хотели его показать. Поэтому, если эта работа выполняется с GDI+, то вы должны протестировать свое приложение в разных цветовых режимах. (Можно также программно включать определенный цветовой режим, хотя мы не обсуждаем это в настоящей главе из-за недостатка места.)

Безопасная палитра

Выше мы уже упомянули вскользь так называемую "безопасную палитру" (safety palette), которая очень часто используется в качестве палитры по умолчанию. Принцип ее организации заключается в том, что установлено шесть равномерно распределенных значений для каждого компонента цвета: 0, 52, 102, 153, 204 и 255. Другими словами, красный компонент может иметь любое из этих значений. То же самое и с зеленым и с синим компонентом.

Таким образом, возможные цвета из безопасной палитры включают (0,0,0) — черный, (153,0,0) — насыщенный темно-красный, (0,255,102) — зеленый с небольшой примесью синего и так далее. Это дает в сумме 6 в кубе, т.е. 216 цветов. Идея в том, чтобы создать палитру, включающую цвета, равномерно распределенные в пределах спектра и всех степеней яркости, хотя на практике это не так хорошо работает, потому что математически равномерное распределение компонентов цветов не означает, что так оно будет воспринято человеческим глазом.

Если переключить Windows в 256-цветный режим, то это даст как раз безопасную палитру с добавочными 20 стандартными цветами Windows и еще 20 запасными.

Перья и кисти

В этом разделе рассматриваются два вспомогательных класса, необходимые для рисования фигур. Мы уже сталкивались с классом `Pen`, который применяли для указания экземпляру `Graphics` способа рисования линий. Связанный с ним класс — `System.Drawing.Brush` — позволяет указать экземпляру `Graphics` способ закрашивания экранных областей. Например, `Pen` необходим был для рисования границ прямоугольника и эллипса в предыдущих примерах. Если бы нам понадобилось нарисовать эти фигуры как сплошные, нужно было бы указать кисть, чтобы специфицировать, как их следует закрашивать. Один общий аспект этих двух классов состоит в том, что вам вряд ли придется когда-нибудь вызывать их методы. Вы просто конструируете эк-

экземпляр Pen или Brush с нужным цветом и другими свойствами, и потом передаете методам рисования, которые их требуют.

Если вы ранее программировали с использованием GDI, то должны были обратить внимание на то, что перья в GDI+ используются несколько иначе. В GDI нормальной практикой было вызывать функцию Windows API SelectObject(), которая в действительности ассоциировала перо с контекстом устройства. Затем это перо применялось во всех операциях рисования, которым оно было необходимо, до тех пор, пока вы не сообщали контексту устройства о том, что нужно использовать другое перо, снова вызывая SelectObject(). Тот же принцип применяется в отношении кистей и других объектов — таких как шрифты и битовые изображения. В GDI+ разработчики Microsoft предпочли модель без сохранения состояния, не предполагающую применение пера или других вспомогательных объектов по умолчанию. Вместо этого при каждом вызове метода передается конкретный вспомогательный объект.

Кисти

GDI+ включает несколько разных видов кистей — больше, чем мы в состоянии описать в настоящей главе, поэтому в данном разделе мы только объясним простейшие из них, чтобы дать вам представление об основных принципах. Каждый тип кисти представлен экземпляром класса, унаследованного от абстрактного класса System.Drawing.Brush. Простейшая кисть, System.Drawing.SolidBrush, означает, что область должна быть заполнена сплошным цветом:

```
Brush solidBeigeBrush = new SolidBrush(Color.Beige);
  Brush solidFunnyOrangyBrownBrush =
       new SolidBrush(Color.FromArgb(255,155,100));
```

В качестве альтернативы, если нужна кисть одного из Web-безопасных цветов, ее можно сконструировать, используя другой класс — System.Drawing.Brushes. Brushes — это один из тех классов, экземпляры которого никогда не создаются (у него есть приватный конструктор, который не позволит это сделать). Он просто включает большое число статических свойств, каждое из которых возвращает кисть заданного цвета. Brushes можно использовать следующим образом:

```
Brush solidAzureBrush = Brushes.Azure;
Brush solidChocolateBrush = Brushes.Chocolate;
```

Следующий уровень сложности представляет штриховая кисть (hatch brush), которая заполняет область, рисуя некоторый шаблон. Тип этой кисти продуман более глубоко, поэтому он находится в пространстве имен Drawing2D и представлен классом System.Drawing.Drawing2D.HatchBrush. Класс Brushes ничем не может помочь в создании штриховых кистей — их нужно конструировать явно, указывая стиль штрихования и два цвета — цвет переднего плана и цвет фона (цвет фона можно опустить, при этом по умолчанию применяется черный). Стиль штрихования выбирается из перечисления System.Drawing.Drawing2D.HatchStyle. Доступен выбор из огромного числа значений HatchStyle (в документации SDK можно увидеть полный список). Чтобы дать общее представление, упомянем часто используемые стили: ForwardDiagonal, Cross, DiagonalCross, SmallConfetti и ZigZag. Ниже показаны примеры конструирования штриховых кистей.

```
Brush crossBrush = new HatchBrush(HatchStyle.Cross, Color.Azure);
// Фоновый цвет для CrossBrush - черный
Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
                       Color.DarkGoldenrod, Color.Cyan);
```

Набор кистей GDI ограничивался только сплошными и штриховыми. GDI+ добавляет к ним еще пару дополнительных кистей:

- `System.Drawing.Drawing2D.LinearGradientBrush` заполняет область цветом, плавно изменяемым по экрану;
- `System.Drawing.Drawing2D.PathGradientBrush` похожа на предыдущую, но здесь цвет меняется вдоль пути вокруг заполняемой области.

Следует отметить, что с помощью последних двух кистей можно создавать впечатляющие эффекты при умелом их использовании.

Перья

В отличие от кистей перья представлены одним классом — `System.Drawing.Pen`. Однако перо — несколько более сложное понятие, нежели кисть, поскольку для него должна указываться толщина линии (в пикселях), а также — для толстых линий — способ заполнения области внутри линии. Для перьев также можно задавать множество других свойств, которые не описаны в настоящей главе, включая упомянутое ранее свойство `Alignment`. Это свойство определяет, где по отношению к границам фигуры должна быть проведена линия, а также какую форму должен иметь ее конец.

Пространство внутри линии может быть залито сплошным цветом либо заполнено с помощью кисти. Поэтому экземпляр `Pen` может содержать ссылку на экземпляр `Brush`. Это довольно-таки мощная технология, поскольку означает, что вы можете рисовать линии, закрашенные, скажем, штриховой или градиентной кистью. Существует четыре разных способа сконструировать экземпляр `Pen` вручную. Можно сделать это, передав цвет либо кисть. Оба эти конструктора создают перо шириной в один пиксель. В качестве альтернативы можно передать цвет или кисть и дополнительно значение типа `float`, представляющее ширину пера (здесь применяется `float` на случай работы `Graphics` в единицах измерения, отличных от пикселей — таких как миллиметры или дюймы). Например, сконструировать перья можно следующим образом:

```
Brush brickBrush = new HatchBrush(HatchStyle.DiagonalBrick,
       Color.DarkGoldenrod, Color.Cyan);
Pen solidBluePen = new Pen(Color.FromArgb(0,0,255));
Pen solidWideBluePen = new Pen(Color.Blue, 4);
Pen brickPen = new Pen(brickBrush);
Pen brickWidePen = new Pen(brickBrush, 10);
```

Вдобавок, для быстрого создания перьев можно применить класс `System.Drawing.Pens`, который, подобно классу `Brushes`, содержит множество готовых перьев. Все эти перья имеют ширину в один пиксель и цвета из числа безопасных к Web. Это позволяет конструировать перья вот так:

```
Pen solidYellowPen = Pens.Yellow;
```

Рисование фигур и линий

Первая часть главы почти завершена, и вы ознакомились со всеми базовыми классами и объектами, необходимыми для рисования на экране фигур и тому подобного. Этот раздел мы начнем с обзора некоторых методов рисования, представленных в классе `Graphics`, а также рассмотрим краткий пример, иллюстрирующий использование нескольких кистей и перьев.

`System.Drawing.Graphics` включает большое количество методов, которые позволяют рисовать разнообразные линии, контурные и сплошные фигуры. Опять-таки отметим, что их слишком много, чтобы можно было привести здесь полный список,

но в табл. 33.4 перечислены некоторые из них, просто чтобы дать вам представление обо всем разнообразии фигур, которые можно нарисовать.

Прежде чем покончить с рисованием простых объектов, в этом разделе мы подведем итог, рассмотрев простой пример, демонстрирующий различные виды визуальных эффектов, которые можно создать с помощью кистей.

Таблица 33.4. Некоторые методы рисования класса `System.Drawing.Graphics`

Метод	Типичные параметры	Что рисует
`DrawLine`	Перо, начальная и конечная точки	Одиночная прямая линия.
`DrawRectangle`	Перо, положение и размер	Контур прямоугольника.
`DrawEllipse`	Перо, положение и размер	Контур эллипса.
`FillRectangle`	Кисть, положение и размер	Сплошной прямоугольник.
`FillEllipse`	Кисть, положение и размер	Сплошной эллипс.
`DrawLines`	Перо, массив точек	Серия линий, соединяющую каждую точку со следующей в массиве.
`DrawBezier`	Перо, четыре точки	Гладкая кривая через две конечные точки с остальными двумя точками, управляющими ее формой.
`DrawCurve`	Перо, массив точек	Гладкая кривая через все точки.
`DrawArc`	Перо, прямоугольник, два угла	Сегмент круга между заданными углами.
`DrawClosedCurve`	Перо, массив точек	Подобно `DrawCurve`, но с прямой линией, замыкающей кривую.
`DrawPie`	Перо, прямоугольник, два угла	Клиноподобный контур внутри прямоугольника.
`FillPie`	Кисть, прямоугольник, два угла	Сплошная клиноподобная область внутри прямоугольника.
`DrawPolygon`	Перо, массив точек	Подобно `DrawLines`, но также с соединением первой и последней точек для замыкания фигуры.

Пример называется `ScrollMoreShapes` и, по сути, является усовершенствованием `ScrollShapes`. Помимо прямоугольника и эллипса мы добавим толстую линию и закрасим фигуры различными кистями. Мы уже изучили базовые принципы рисования, поэтому код будет говорить сам за себя. Во-первых, поскольку будут применены новые кисти, понадобится включить пространство имен `System.Drawing.Drawing2D`:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Text;
using System.Windows.Forms;
```

Далее добавим в наш класс `Form1` несколько дополнительных полей, которые будут хранить информацию о месте, где должны быть нарисованы фигуры, а также об используемых перьях и кистях:

```
private Rectangle rectangleBounds = new Rectangle(new Point(0,0),
                                    new Size(200,200));
private Rectangle ellipseBounds = new Rectangle(new Point(50,200),
                                    new Size(200,150));
private readonly Pen bluePen = new Pen(Color.Blue, 3);
private readonly Pen redPen = new Pen(Color.Red, 2);
private readonly Brush solidAzureBrush = Brushes.Azure;
private readonly Brush solidYellowBrush = new SolidBrush(Color.Yellow);
private static readonly Brush brickBrush =
  new HatchBrush(HatchStyle.DiagonalBrick, Color.DarkGoldenrod, Color.Cyan);
private Pen brickWidePen = new Pen(brickBrush, 10);
```

Поле `brickBrush` объявлено статическим, чтобы можно было использовать его значение для инициализации поля `brickWidePen`. Язык C# не позволяет использовать одно поле экземпляра для инициализации другого поля экземпляра, поскольку не определено, какое из них инициализируется первым. Однако объявление одного из полей статическим решает проблему. А так как у нас создается только один экземпляр класса `Form1`, то неважно, является поле статическим или нет.

Переопределим `OnPaint()` следующим образом:

```
protected override void OnPaint( PaintEventArgs e )
{
  base.OnPaint(e);
  Graphics dc = e.Graphics;
  Point scrollOffset = AutoScrollPosition;
  dc.TranslateTransform(scrollOffset.X, scrollOffset.Y);
  if (e.ClipRectangle.Top+scrollOffset.X < 350 ||
     e.ClipRectangle.Left+scrollOffset.Y < 250)
  {
     dc.DrawRectangle(bluePen, rectangleBounds);
     dc.FillRectangle(solidYellowBrush, rectangleBounds);
     dc.DrawEllipse(redPen, ellipseBounds);
     dc.FillEllipse(solidAzureBrush, ellipseBounds);
     dc.DrawLine(brickWidePen, rectangleBounds.Location,
             ellipseBounds.Location+ellipseBounds.Size);
  }
}
```

Как и раньше, установим `AutoScrollMinSize` равным (250,350). На рис. 33.13 показан результат.

Обратите внимание, что толстая диагональная линия нарисована поверх прямоугольника и эллипса, потому что она нарисована последней.

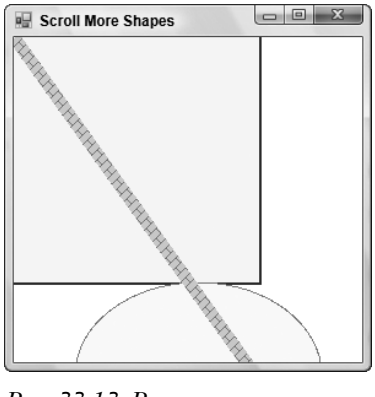

Рис. 33.13. Результат выполнения примера `ScrollMoreShapes`

Вывод графических изображений

Одной из операций, которые вам, вероятно, придется делать с помощью GDI+, будет вывод готовых изображений, хранящихся в файлах. В действительности это намного проще, чем рисование собственного пользовательского интерфейса, потому что в данном случае все уже нарисовано заранее. По сути все, что нужно сделать — это загрузить файл и дать команду GDI+ отобразить его. Изображение может представлять простой рисунок, состоящий из линий, пиктограмму или же сложную картинку наподобие фотографии. Изображением можно также манипулировать, растягивая и поворачивая его или же отображая какую-то его часть.

Начнем этот раздел с простого примера. Затем обсудим некоторые обстоятельства, о которых необходимо помнить при отображении картинок. Такой путь возможен, поскольку код, визуализирующий графические изображения, очень прост.

Класс, который нам понадобится — `System.Drawing.Image` — это один из базовых классов .NET. Экземпляр `Image` представляет изображение. Его чтение из файла требует написания всего одной строки кода:

```
Image myImage = Image.FromFile("FileName");
```

`FromFile()`, статический член класса `Image`, является обычным способом создания изображения. Файл может иметь любой из поддерживаемых графических форматов, включая `.bmp`, `.jpg`, `.gif` и `.png`.

Вывод изображения также очень прост, если у вас в руках уже есть подходящий экземпляр `Graphics`. Для этого достаточно вызова `Graphics.DrawImageUnscaled()` или `Graphics.DrawImage()`. Доступно сравнительно немного перегрузок этих методов, но они обеспечивают существенную гибкость за счет указания информации относительно местоположения и размера отображаемого образа. Но в данном примере мы используем `DrawImage()` следующим образом:

```
dc.DrawImage(myImage, points);
```

В этой строке кода предполагается, что `dc` — экземпляр `Graphics`, а `myImage` — отображаемый экземпляр `Image`. Аргумент `points` — массив структур `Point`, где `points[0]`, `points[1]` и `points[2]` — координаты верхнего левого, верхнего правого и нижнего левого углов картинки.

Работа с графическими изображениями — возможно, та область, в которой разработчики, знакомые с GDI, обнаружат наибольшие различия между GDI и GDI+. В GDI вывод изображения требовал выполнения нескольких нетривиальных шагов. Если изображение представляло собой битовую карту, его загрузка была относительно простой. Но если это был файл другого типа, то его загрузка требовала последовательности вызовов OLE-объектов. Вывод загруженного изображения на экран включал в себя получение его дескриптора, выбор его в контексте устройства, находящегося в памяти, и выполнение блочной передачи между контекстами. Хотя контексты устройств и дескрипторы остались "за кулисами" и по-прежнему необходимы в случае программного редактирования образа, простые задачи теперь исключительно удобно помещены в оболочки объектной модели GDI+.

Проиллюстрируем процесс вывода графического изображения на примере под названием `DisplayImage`. Пример будет просто отображать файл `.jpg` в главном окне приложения. Для простоты путевое имя файла `.jpg` жестко закодировано в тексте программы (поэтому перед запуском откорректируйте это имя в соответствии с местоположением файла в вашей системе). Отображаемый файл `.jpg` будет фотографией заката над Санкт-Петербургом.

Как и другие примеры, проект `DisplayImage` — стандартный проект приложения Windows на C#, сгенерированного Visual Studio 2008. Добавим в класс `Form1` следующие поля:

```
readonly Image piccy;
private readonly Point [] piccyBounds;
```

В конструкторе `Form1` загрузим файл с картинкой:

```
public Form1()
{
     InitializeComponent();
     piccy = Image.FromFile(@"C:\ProCSharp\GdiPlus\Images\London.jpg");
     AutoScrollMinSize = piccy.Size;
     piccyBounds = new Point[3];
```

```
        piccyBounds[0] = new Point(0,0);                    // верхний левый
        piccyBounds[1] = new Point(piccy.Width,0);          // верхний правый
        piccyBounds[2] = new Point(0,piccy.Height);         // нижний левый
    }
```

Обратите внимание, что размер изображения в пикселях получается из его свойства `Size` и используется для установки области документа. Также создается массив точек `piccyBounds`, определяющий позицию изображения на экране. Мы устанавливаем координаты точек в соответствии с действительным размером картинки, но при желании ее можно было бы увеличить, уменьшить, растянуть или даже втиснуть в непрямоугольный параллелограмм, всего лишь изменив значения структур `Point` в массиве `piccyBounds`.

Картинка отображается в переопределенном методе `OnPaint()`:

```
protected override void OnPaint(PaintEventArgs e)
{
     base.OnPaint(e);
     Graphics dc = e.Graphics;
     dc.ScaleTransform(1.0f, 1.0f);
     dc.TranslateTransform(AutoScrollPosition.X, AutoScrollPosition.Y);
     dc.DrawImage(piccy, piccyBounds);
}
```

И, наконец, внесем изменение в сгенерированный IDE код метода `Form1.Dispose()`:

```
protected override void Dispose(bool disposing)
{
   piccy.Dispose();
   if( disposing && (component != null))
   {
      components.Dispose();
   }
   base.Dispose(disposing );
}
```

Уничтожить изображение сразу после того, как отпадает в нем необходимость, важно потому, что обычно графические изображения занимают большие объемы памяти. После того, как вызван `Image.Dispose()`, экземпляр `Image` более не ссылается ни на какое реальное изображение, а потому не может его отображать (если только не загружено новое изображение). На рис. 33.14 показан результат выполнения этого кода.

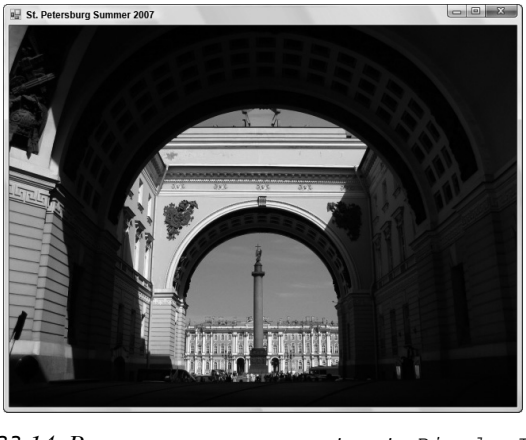

Рис. 33.14. Результат выполнения примера `DisplayImage`

Вопросы манипулирования изображениями

Хотя вывод графических изображений очень прост, все же он требует некоторого понимания того, что при этом происходит "за кулисами".

Наиболее важный для понимания момент насчет изображений — они всегда прямоугольны. Это не просто соглашение, это следствие лежащей в основе технологии. Все современные графические карты так устроены, что могут эффективно копировать блоки пикселей из одной области памяти в другую, только если эти блоки пикселей описывают прямоугольную область. Эта ускоренная аппаратно операция может виртуально выглядеть как атомарная, а потому выполняться чрезвычайно быстро. На самом деле, это — ключ к современной высокопроизводительной графике. Эта операция называется *передачей битового блока* (bitmap block transfer, или *BitBlt*).

Метод `Graphics.DrawImageUnscaled()` внутри использует `BitBlt`, что позволяет нам видеть огромные изображения, содержащие, возможно, миллионы пикселей, которые появляются почти мгновенно. Если бы компьютеру приходилось копировать изображение на экран пиксель за пикселем, то мы бы увидели, как он появляется постепенно, в течение периода длительностью до нескольких секунд.

`BitBlt` — чрезвычайно эффективная операция, а потому почти все, что связано с отображением картинок и манипулировании ими, выполняется с ее помощью. Даже программное редактирование графических изображений, предусматривающее манипулирование их частями, осуществляется с помощью операции `BitBlt` между контекстами устройств, представленными областями памяти. Во времена GDI функция Windows API `BitBlt()` была, наверно, самой важной и наиболее широко используемой функцией для манипуляции изображениями, хотя в GDI+ операция `BitBlt` в большой степени скрыта в объектной модели GDI+.

Невозможно выполнить операцию `BitBlt` над непрямоугольными областями, хотя этот эффект можно легко эмулировать. Один из способов является пометка определенного цвета как прозрачного для `BitBlt`, так что области, окрашенные этим цветом, не перекроют существующие пиксели на целевом устройстве. Можно также указать, что в процессе выполнения `BitBlt` каждый пиксель результирующего изображения должен быть сформирован в результате некоторой логической операции (такой как битовое "И") над цветами пикселей исходного и целевого изображений. Такие операции поддерживаются аппаратными акселераторами и могут быть использованы для получения разнообразных тонких эффектов. Отметим также, что в объекте `Graphics` также реализован и другой метод — `DrawImage()`. Он подобен `DrawImageUnscaled()`, но представлен большим количеством перегрузок, которые позволяют специфицировать более сложные формы `BitBlt`, используемые в процессе отображения. `DrawImage()` также позволяет рисовать (с помощью `BitBlt`) только определенную часть изображения либо выполнять некоторые операции, такие как масштабирование (увеличивая или уменьшая размер).

Рисование текста

Раскрытие очень важной темы отображения текста в этой главе было отложено вплоть до настоящего момента, поскольку рисование текста на экране (в общем случае) более сложно, чем рисование простой графики. Хотя чрезвычайно легко вывести одну или две строки текста, не особенно заботясь об их внешнем виде — для этого нужен один простой вызов метода `Graphics.DrawString()` — но если вы попытаетесь отобразить документ, который включает в себя значительный объем текста, то быстро обнаружите, что все существенно усложнится. На то есть две причины.

- Если вы хотите, чтобы текст выглядел привлекательно, вам следует разобраться с концепцией шрифтов. В то время как для рисования фигур необходимы кисти и перья в качестве вспомогательных объектов, процесс рисования текста требует в качестве вспомогательных объектов применять шрифты. И понимание шрифтов — непростая задача.
- Текст должен быть размещен в окне очень тщательно. Обычно пользователи ожидают, что слова будут следовать одно за другим, и четко выровнены. Обеспечить это сложнее, чем может показаться на первый взгляд. Для начала, скажем, что обычно вы не можете предвидеть, сколько места на экране должно занять то или иное слово. Это нужно вычислять (используя метод `Graphics.MeasureString()`). К тому же пространство, занятое словом на экране, зависит от его относительного положения в документе. Если ваше приложение должно выполнять перенос строк, то вам придется особенно тщательно рассчитывать размеры слов, чтобы принять решение, где следует поместить перенос строки. Когда вы в следующий раз запустите Microsoft Word, обратите внимание на то, как Word постоянно переупорядочивает текст по мере его ввода: ему приходится выполнять огромный объем сложной обработки. Конечно, вряд ли какое-то из ваших приложений GDI+ будет настолько сложным, как Word. Однако если вам необходимо отображать любой текст, то вам придется решать многие из аналогичных задач.

Короче говоря, корректную высококачественную обработку текста реализовать нелегко. Однако очень просто поместить строку текста на экран, когда известен шрифт и место, где она должна появиться. Поэтому в следующем разделе мы рассмотрим простейший пример, демонстрирующий отображение некоторого текста, за которым последует краткий обзор принципов построения шрифтов и их семейств, а потом — более реалистичный (и сложный) пример приложения, обрабатывающего тексты — `CapsEditor`.

Простой пример отображения текста

Этот пример, `DisplayText`, снова будет обычным приложением Windows Forms. На этот раз мы переопределим метод `OnPaint()` и добавим поля-члены:

```
private readonly Brush blackBrush = Brushes.Black;
private readonly Brush blueBrush = Brushes.Blue;
private readonly Font haettenschweilerFont = new Font("Haettenschweiler", 12);
private readonly Font boldTimesFont = new Font("Times New Roman", 10, FontStyle.Bold);
private readonly Font italicCourierFont = new Font("Courier", 11, FontStyle.Italic |
     FontStyle.Underline);
protected override void OnPaint(PaintEventArgs e)
{
  base.OnPaint(e);
  Graphics dc = e.Graphics;
  dc.DrawString("This is a groovy string", haettenschweilerFont, blackBrush,
                10, 10);
  dc.DrawString("This is a groovy string " +
                "with some very long text that will never fit in the box",
                boldTimesFont, blueBrush,
                new Rectangle(new Point(10, 40), new Size(100, 40)));
  dc.DrawString("This is a groovy string", italicCourierFont, blackBrush,
                new Point(10, 100));
}
```

На рис. 33.15 показан результат выполнения этого примера.

Данный пример демонстрирует применение метода `Graphics.DrawString()` для отображения текста. Метод `DrawString()` имеет множество перегрузок, три из которых продемонстрированы в коде. Различные перегрузки требуют параметров, указывающих отображаемый текст, шрифт, в котором его следует отобразить, а также кисть, применяемую для конструирования прямых и кривых линий, составляющих каждый символ текста. Для остальных параметров существует несколько альтернативных вариантов. Однако, как правило, имеется также возможность указать `Point` (или два числа) или `Rectangle`.

Рис. 33.15. Результат выполнения примера `DisplayText`

Если вы указываете `Point`, то левый верхний угол текста размещается в этой точке и просто распространяется от нее вправо и вниз. Если же вы указываете `Rectangle`, то в этом случае экземпляр `Graphics` размещает текст внутри указанно прямоугольника. Если же текст не умещается в пределы этого прямоугольника, он будет усечен (см. четвертую строку текста на рис. 33.15). Передача методу `DrawString()` прямоугольника означает, что процесс рисования будет длиться дольше, потому что `DrawString()` придется определять места переноса строк. Однако результат выглядит лучше, если строки умещаются в прямоугольник.

В этом примере также показано множество способов конструирования шрифтов. Всегда необходимо указать имя шрифта и его размер (высоту). Кроме того, можно необязательно передать различные стили, модифицирующие отображение текста (жирный, подчеркнутый, курсив и тому подобное).

Шрифты и их семейства

Шрифт описывает, как именно должна отображаться каждая буква. Выбор соответствующего шрифта и представление разумного их разнообразия в пределах документа — важные факторы обеспечения его читаемости.

Большинство людей, когда их просят назвать какой-нибудь шрифт, могут вспомнить Arial или Times New Roman (если они работают в Windows) либо Times или Helvetica (если они пользователи Mac OS). На самом деле это вовсе не шрифты. Это — семейства шрифтов. Семейство шрифтов определяет в общих терминах визуальный стиль текста и является ключевым фактором общей привлекательности вашего приложения. Большинство из нас легко распознают стили наиболее распространенных семейств шрифтов, даже если не задумываются о них.

Действительный *шрифт* — это нечто вроде "Arial 9-point italic". Другими словами, для конкретного шрифта наряду с именем семейства указывается размер и другие модифицирующие параметры текста. Эти модификации могут описывать **полужирный**, *курсив*, подчеркнутый, КАПИТЕЛЬ или $_{\text{подстрочный}}$; это то, что часто называют *стилем*, хотя в некотором смысле этот термин вводит в заблуждение, потому что визуальное представление в большей степени определяется семейством шрифтов.

Размер шрифта определяется его высотой. Высота измеряется в *точках* (points) — традиционных единицах измерения, представляющих 1/72 часть дюйма (0,351 мм). То есть 10-точечный шрифт имеет высоту 1/7 дюйма, или приблизительно 3,5 мм. Однако это не означает, что семь строк 10-точечного шрифта уместятся в один дюйм по вертикали на экране или бумаге, потому что между строками должно быть некоторое пространство.

Строго говоря, измерение высоты шрифта не так-то просто, потому что существует несколько разных высот, которые следует учитывать. Например, есть высота заглавных букв, таких как A или F (именно об их высоте идет речь, когда говорят о высоте шрифта), есть дополнительная высота, занятая знаками акцентирования в таких буквах, как Å или Ñ, а также дополнительная высота под базовой линией, необходимая для отображения букв вроде y или q. Однако в этой главе мы не будем обращать внимание на это. Когда вы указываете семейство шрифтов и основную высоту, то все остальные высоты определяются автоматически.

Имея дело со шрифтами, можно столкнуться также с другими часто используемыми терминами, описывающими определенные их семейства.

- **Serif** — это семейство шрифтов характеризуется наличием на многих кончиках букв небольших засечек (эти засечки и называются "serif"). Классический пример такого шрифта — Times New Roman.
- **Sans serif** — в отличие от предыдущего семейства, засечек не имеет. Хорошим примером могут служить шрифты Arial или Verdana. Отсутствие засечек часто делает текст резким, бросающимся в глаза, поэтому часто такие шрифты используются для выделения наиболее важного текста.
- **True Type** — семейство шрифтов, символы которых точно описываются математически как состоящие из отрезков прямых и кривых линий. Это значит, что одно и то же определение может быть использовано для вычисления отображения шрифтов любого размера в пределах семейства. В наше время почти все шрифты, которые вы можете применять, относятся к этой группе. Только некоторые старые семейства шрифтов, унаследованные еще от Windows 3.1, определены как отдельные битовые изображения для каждого символа каждого из возможных размеров. Использовать их в настоящее время не рекомендуется.

Для выбора и манипуляций шрифтами Microsoft предоставляет два основных класса:

- `System.Drawing.Font`
- `System.Drawing.FontFamily`

Вы уже видели основное применение класса `Font`. Когда требуется нарисовать текст, создается экземпляр `Font` и передается методу `DrawString()` для указания того, как он должен быть нарисован. Экземпляр `FontFamily` используется для представления семейства шрифтов.

Один из случаев применения класса `FontFamily` — это когда вам известно, какой нужен шрифт определенного типа (Serif, Sans Serif или Monospace), но не известно конкретное семейство. Статические свойства `GenericSerif`, `GenericSansSerif` и `GenericMonospace` возвращают шрифты по умолчанию, удовлетворяющие заданным критериям:

```
FontFamily sansSerifFont = FontFamily.GenericSansSerif;
```

Вообще говоря, если вы разрабатываете профессиональное приложение, то вам следует выбирать шрифты более изощренным способом. Скорее всего, вы реализуете код рисования так, чтобы он проверял доступные семейства шрифтов и выбирал из них наиболее подходящие — возможно, выбирая первый доступный из списка предпочитаемых. И если вы хотите, чтобы ваше приложение было дружественным к пользователю, то первым выбором в списке должен быть тот шрифт, который пользователь выбирал во время предыдущего запуска приложения. Обычно, если вы имеете дело с наиболее популярными семействами шрифтов, такими как Arial и Times New Roman, то вы — в безопасности. Однако если вы пытаетесь отобразить текст, используя не-

существующие шрифты, результат не всегда предсказуем, и, скорее всего, вы обнаружите, что операционная система Windows самостоятельно подставляет вместо них стандартный системный шрифт, который ей очень легко отображать, но который выглядит не особенно привлекательно — и когда он появится в документе, у пользователя возникнет впечатление, что он имеет дело с программным обеспечением низкого качества.

Для того чтобы получить список доступных в системе шрифтов, необходимо воспользоваться классом `InstalledFontCollection`, определенным в пространстве имен `System.Drawing.Text`. Этот класс реализует свойство `Families`, представляющее собой массив всех шрифтов, доступных в системе:

```
InstalledFontCollection insFont = new InstalledFontCollection();
FontFamily [] families = insFont.Families;
foreach (FontFamily family in families)
{
  // выполнить обработку с данным семейством шрифтов
}
```

Пример: перечисление семейств шрифтов

В этом разделе мы разработаем небольшой пример `EnumFontFamilies`, который перечислит все семейства шрифтов, доступные в системе, и проиллюстрирует их, отображая имя каждого семейства с помощью соответствующего шрифта (в его стандартной 10-точечной версии). На рис. 33.16 показан результат выполнения этого примера.

Рис 33.16. Результат выполнения примера `EnumFontFamilies`

Естественно, конкретный результат, который вы получите на своем компьютере, зависит от перечня установленных у вас шрифтов.

Как обычно, для этого примера создадим Windows-приложение на C# под названием `EnumFontFamilies`. Начнем с добавления ссылки на необходимое пространство имен. Мы будем использовать класс `InstalledFontCollection`, который определен в `System.Drawing.Text`:

```
using System.Drawing;
using System.Drawing.Text;
using System.Windows.Forms;
```

Затем добавим следующую константу в класс `Form1`:

```
private const int margin = 10;
```

`margin` — это размер левого и верхнего отступов текста от края документа; он предотвращает появление текста прямо на границе клиентской области.

Этот пример предназначен для простого и быстрого показа наименований семейств шрифтов, поэтому код его достаточно сырой и во многих отношениях работает не так, как должно работать реальное приложение. Например, здесь мы жестко кодируем предполагаемый размер документа (200,1500) и устанавливаем это значение свойству `AutoScrollMinSize` в окне свойств интегрированной среды разработки Visual Studio 2008. По большому счету следовало бы проверить размер отображаемого текста, чтобы определить реальный размер документа. Это мы сделаем в следующем разделе.

Вот метод `OnPaint()`:

```
protected override void OnPaint(PaintEventArgs e)
{
  base.OnPaint(e);
  int verticalCoordinate = margin;
  Point topLeftCorner;
  InstalledFontCollection insFont = new InstalledFontCollection();
  FontFamily [] families = insFont.Families;
  e.Graphics.TranslateTransform(AutoScrollPosition.X,
                                AutoScrollPosition.Y);
  foreach (FontFamily family in families)
  {
     if (family.IsStyleAvailable(FontStyle.Regular))
     {
       Font f = new Font(family.Name, 12);
       Point topLeftCorner = new Point(margin, verticalCoordinate);
       verticalCoordinate += f.Height;
       e.Graphics.DrawString (family.Name, f, Brushes.Black,topLeftCorner);
       f.Dispose();
     }
  }
}
```

Код начинается с использования объекта `InstalledFontCollection` для получения массива, содержащего всю информацию относительно доступных семейств шрифтов. Для каждого семейства создается экземпляр 12-точечного шрифта. Для этого применяется самый простой конструктор `Font` (существует множество других с дополнительными опциями). Этот конструктор принимает два параметра — имя семейства и размер шрифта:

```
Font f = new Font(family.Name, 12);
```

Конструктор создает шрифт стандартного стиля. Чтобы подстраховаться, однако, сначала мы проверяем, существует ли шрифт такого стиля в каждом из семейств, прежде чем пытаться что-то выводить с помощью этого шрифта. Проверка осуществляется с помощью метода `FontFamily.IsStyleAvailable()`. Проверка важна, потому что не все шрифты представлены во всех стилях:

```
if (family.IsStyleAvailable(FontStyle.Regular))
```

`FontFamily.IsStyleAvailable()` принимает один параметр — перечисление `FontStyle`. Перечисление содержит множество флагов, которые могут быть скомбинированы битовой операцией "ИЛИ". Возможные флаги таковы: `Bold`, `Italic`, `Regular`, `Strikeout` и `Underline`.

И, наконец, обратите внимание на использование свойства `Height` класса `Font`, которое возвращает высоту отображения текста в данном шрифте, чтобы определить пропуск между строками:

```
Font f = new Font(family.Name, 12);
Point topLeftCorner = new Point(margin, verticalCoordinate);
verticalCoordinate += f.Height;
```

Опять-таки, для простоты в этой версии `OnPaint()` допущены некоторые примеры плохого стиля программирования. Например, мы не заботимся о проверке того, какая именно часть документа нуждается в перерисовке — мы просто отображаем все подряд. Вдобавок к этому создание экземпляра `Font`, как отмечалось ранее — достаточно дорогостоящий по используемым ресурсам процесс, поэтому нам стоило бы сохранять шрифты, а не создавать их новые копии при каждом вызове `OnPaint()`. В результате всех этих небрежностей можно заметить, что данный пример приложения требует заметного времени для того, чтобы перерисовать себя. Дабы сэкономить

память и облегчить задачу сборщику мусора, мы вызываем `Dispose()` после завершения работы с каждым созданным экземпляром шрифта. Если этого не делать, то после 10 или 20 вызовов операций перерисовки окажется израсходованным огромное количество памяти для сохранения ненужных более шрифтов.

Редактирование текстового документа: пример `CapsEditor`

Теперь мы подошли к наиболее расширенному примеру настоящей главы. Пример `CapsEditor` предназначен для демонстрации того, как принципы рисования, которые мы изучили до сих пор, можно применить в более реалистичном контексте. Пример `CapsEditor` не потребует изучения никакого нового материала, помимо реагирования на пользовательский ввод мышью, но покажет, как управлять отображением текста таким образом, чтобы обеспечить достаточную производительность приложения, в то же время гарантируя постоянное обновление содержимого клиентской области главного окна.

Программа `CapsEditor` дает возможность пользователю читать текстовый файл, который затем отображается строка за строкой в текстовой области. Если пользователь выполнит двойной щелчок на строке, строка преобразуется к верхнему регистру. Это, по сути, все. Но даже несмотря на столь ограниченный набор возможностей, вы увидите, что работа, которую необходимо выполнить для того, чтобы обеспечить правильное отображение текста в правильном месте при гарантии нормальной производительности — достаточно сложная задача. В частности, здесь появится новый элемент: содержимое документа может изменяться — либо при выборе команды меню для чтения нового файла, либо при двойном щелчке на строке. В первом случае потребуется обновить размер документа, чтобы линейки прокрутки работали правильно, и затем перерисовать все заново. Во втором случае необходимо тщательно проверить, изменился ли размер документа, и какой текст должен быть повторно отображен.

Этот раздел мы начнем с внешнего вида `CapsEditor`. Когда приложение впервые запускается, оно не имеет загруженного документа и выглядит, как показано на рис. 33.17.

Меню File (Файл) включает две команды: Open (Открыть), которая вызывает диалоговое окно `OpenFileDialog` и читает выбранный в нем файл, а также Exit (Выход), закрывающая приложение. На рис. 33.18 показана программа `CapsEditor`, отображающая свой собственный исходный файл, `Form1.cs` (с несколькими строками, переведенными в верхний регистр в результате двойных щелчков на них).

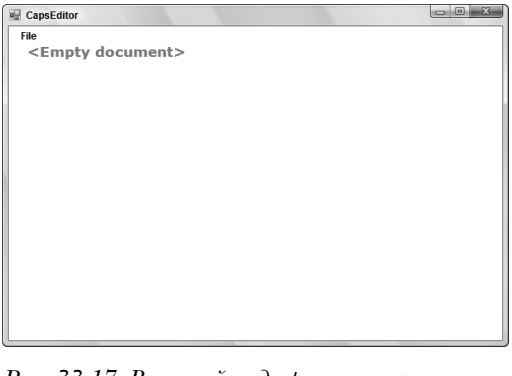

Рис. 33.17. Внешний вид приложения `CapsEditor` после первого запуска

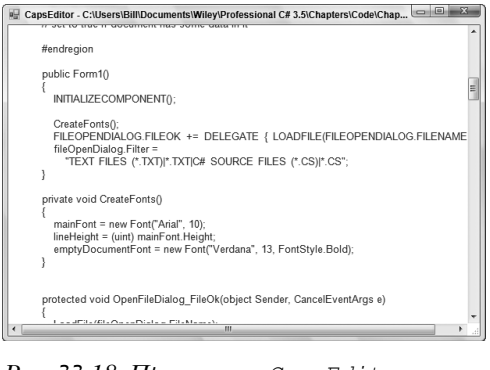

Рис. 33.18. Приложение `CapsEditor` отображает содержимое файла `Form1.cs`

Размеры вертикальной и горизонтальной линеек прокрутки корректны. Клиентская область прокручивается ровно настолько, чтобы можно было увидеть весь документ. CapsEditor не пытается переносить строки текста — пример и без того достаточно сложен. Он просто отображает каждую строку текста в точности, как она была прочитана. Никаких ограничений на размер файла не накладывается, но предполагается, что это текстовый файл, который не содержит в себе никаких непечатаемых символов. Начнем с добавления оператора using:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Drawing;
using System.IO;
using System.Windows.Forms;
```

Это потому, что мы будем использовать класс StreamReader, находящийся в пространстве имен System.IO. Добавим некоторые новые поля в класс Form1:

```
#region Constant fields
private const string standardTitle = "CapsEditor"; // текст заголовка по умолчанию
private const uint margin = 10; // горизонтальное и вертикальное
                                // поля отступов клиентской области
#endregion
#region Member fields
// 'документ'
private readonly List<TextLineInformation> documentLines = new
  List<TextLineInformation> ();
private uint lineHeight;    // высота одной строки в пикселях
private Size documentSize;  // необходимый размер клиентской области
                            // для отображения всего документа
private uint nLines;        // количество строк в документе
private Font mainFont;      // шрифт, используемый для отображения строк документа
private Font emptyDocumentFont;  // шрифт, используемый для отображения
                           // пустого сообщения
private readonly Brush mainBrush = Brushes.Blue; // кисть, используемая для
                                                 // отображения текста документа
private readonly Brush emptyDocumentBrush = Brushes.Red;
               // кисть, используемая для отображения пустого сообщения
private Point mouseDoubleClickPosition; //положение указателя мыши при двойном щелчке
private readonly OpenFileDialog fileOpenDialog = new OpenFileDialog();
               // стандартный диалог открытия файла
private bool documentHasData = false;   // устанавливается true, если документ
содержит данные
#endregion
```

Большинство этих полей не требуют пояснений. Поле documentLines — List<TextLineInformation>, содержащий текст прочитанного файла. В действительности это поле, которое хранит содержимое документа.

Каждый элемент documentLines содержит одну строку текста. Поскольку это поле относится к типу List<TextLineInformation>, а не к простому массиву, в него можно динамически добавлять новые элементы в процессе чтения файла.

Как уже упоминалось ранее, каждый элемент documentLines содержит информацию об одной строке текста. Сама эта информация представлена экземпляром другого класса — TextLineInformation:

```
class TextLineInformation
{
  public string Text;
  public uint Width;
}
```

`TextLineInformation` выглядит классическим случаем, более подходящим для использования структуры вместо класса, потому что это всего лишь пара полей.

Однако обращение к его экземплярам всегда выполняется через элементы `List<TextLineInformation>`, что предполагает их сохранение в виде ссылочных типов.

Каждый экземпляр `TextLineInformation` сохраняет строку текста, а потому представляет минимальный элемент, отображаемый как единое целое. В общем случае для каждого подобного элемента в GDI+ вы, вероятно, захотите хранить вместе с текстом его мировые координаты, указывающие место, где он должен быть отображен, а также его размер (страничные координаты часто изменяются, когда пользователь прокручивает документ, в то время как мировые остаются неизменными до тех пор, пока другие части документа не будут каким-то образом модифицированы). Но в данном случае помимо текста мы сохраняем только ширину (`Width`) элемента. Причина в том, что высота в данном случае определяется выбранным шрифтом. Она остается неизменной для всех строк текста, а потому хранить ее отдельно для каждой строки незачем. Мы сохраняем ее один раз — в поле `Form1.lineHeight`. Что же касается позиции, то в данном случае координата x равна величине поля отступа, а координата y легко вычисляется:

```
margin + lineHeight*(количество строк, предшествующих данной)
```

Если бы мы попытались отображать и манипулировать, скажем, отдельными словами вместо целых строк, то позицию x каждого слова пришлось бы вычислять, используя сумму ширин всех предыдущих слов данной строки. Однако в целях простоты здесь мы ограничимся трактовкой каждой строки текста как единого целого.

Теперь обратимся к главному меню. Эта часть приложения в большей степени относится к миру Windows Forms (см. главу 31), чем к GDI+. Мы добавим команды меню, используя представление конструктора в Visual Studio 2008, но переименуем их в `menuFile`, `menuFileOpen` и `menuFileExit`. Далее добавим обработчики команд меню File⇨Open (Файл⇨Открыть) и File⇨Exit (Файл⇨Выход), используя окно свойств Visual Studio 2008. Эти обработчики событий имеют имена, сгенерированные Visual Studio 2008 — `menuFileOpen_Click()` и `menuFileExit_Click()`.

Добавим дополнительный код инициализации к конструктору `Form1`:

```
public Form1()
{
  InitializeComponent();
  CreateFonts();
  fileOpenDialog.FileOk += delegate { LoadFile(fileOpenDialog.FileName); };
  fileOpenDialog.Filter = "Text files (*.txt)|*.txt|C# source files (*.cs)|*.cs";
}
```

Здесь добавлен обработчик события щелчка для экземпляров, которое возникает, когда пользователь щелкает на кнопке OK диалогового окна File Open (Открытие файла). Мы также устанавливаем фильтр, чтобы можно было загружать только текстовые файлы, а именно — файлы с расширениями `.txt` и файлы исходного кода на C#, поэтому данное приложение можно использовать для просмотра исходного кода самого примера.

`CreateFonts()` — вспомогательный метод, который разбирается с используемыми шрифтами:

```
private void CreateFonts()
{
  mainFont = new Font("Arial", 10);
  lineHeight = (uint)mainFont.Height;
  emptyDocumentFont = new Font("Verdana", 13, FontStyle.Bold);
}
```

Определения обработчиков достаточно стандартны:

```
protected void OpenFileDialog_FileOk(object Sender, CancelEventArgs e)
{
  LoadFile(fileOpenDialog.FileName);
}


protected void menuFileOpen_Click(object sender, EventArgs e)
{
  fileOpenDialog.ShowDialog();
}

protected void menuFileExit_Click(object sender, EventArgs e)
{
  Close();
}
```

Далее рассмотрим метод `LoadFile()`. Он обрабатывает открытие и чтение файла (а также обеспечивает возбуждение события `Paint` для принудительной перерисовки с новым файлом).

```
private void LoadFile(string FileName)
{
  StreamReader sr = new StreamReader(FileName);
  string nextLine;
  documentLines.Clear();
  nLines = 0;
  TextLineInformation nextLineInfo;

  while ( (nextLine = sr.ReadLine()) != null)
  {
     nextLineInfo = new TextLineInformation();
     nextLineInfo.Text = nextLine;
     documentLines.Add(nextLineInfo);
     ++nLines;
  }

  sr.Close();
  documentHasData = (nLines>0) ? true : false;
  CalculateLineWidths();
  CalculateDocumentSize();
  Text = standardTitle + " - " + FileName;
  Invalidate();
}
```

Бо́льшая часть этой функции представляет собой просто стандартный код чтения файла (см. главу 25). Отметим, что после того, как файл открыт, мы последовательно добавляем его строки в `ArrayList` по имени `documentLines`, поэтому в итоге этот массив содержит все строки файла в том порядке, как они были прочитаны. После считывания файла устанавливается флаг `documentHasData`, который говорит о том, есть ли в документе содержимое, подлежащее отображению. Следующая задача — вычислить, где именно все должно отображаться, а, покончив с этим, потребуется определить, какой размер клиентской области понадобится для отображения всего файла — т.е. размер документа, необходимый для настройки линеек прокрутки. И, наконец, устанавливается текст заголовка и вызывается `Invalidate()`. Это один из важнейших методов, предоставленных Microsoft, поэтому в следующем разделе мы первым делом поговорим о нем, прежде чем рассматривать код методов `CalculateLineWidths()` и `CalculateDocumentSize()`.

Invalidate()

`Invalidate()` — член класса `System.Windows.Forms.Form`. Он помечает клиентскую область окна как недействительную, а потому подлежащую перерисовке, и затем убеждается в том, что было возбуждено событие `Paint`. Метод `Invalidate()` имеет пару перегрузок: можно передать ему прямоугольник, указывающий точно (в страничных координатах) позицию и размер перерисовываемой области, или же, если не передавать никаких параметров, то недействительной помечается вся клиентская область.

Если нам известно, что какая-то часть требует перерисовки, почему нельзя просто вызвать `OnPaint()` либо какой-то другой метод, который выполнит ее непосредственно? Ответ состоит в том, что вообще прямой вызов процедур рисования считается дурным тоном программирования. Если ваш код решает что-то перерисовать, он должен вызвать `Invalidate()`. Ниже описаны причины.

- Перерисовка — это почти всегда наиболее ресурсоемкая задача приложения GDI+, а потому выполнение ее посреди другой работы задерживает ее ход. В данном примере, если бы мы напрямую вызвали метод, выполняющий перерисовку из `LoadFile()`, это означало бы, что `LoadFile()` не вернет управление до тех пор, пока не завершится рисование. В это время наше приложение не могло бы реагировать ни на какие другие события. С другой стороны, вызов `Invalidate()` просто заставляет Windows возбудить событие `Paint` перед тем, как немедленно вернуть управление из `LoadFile()`. После этого Windows может свободно просматривать события, подлежащие обработке. Внутреннее устройство этого механизма таково, что события в виде *сообщений* находятся в специальной *очереди сообщений*. Windows периодически просматривает эту очередь, и если в ней есть сообщения, выбирает одно из них и вызывает соответствующий обработчик события. Хотя событие `Paint` может быть единственным, находящимся в очереди (поэтому `OnPaint()` будет вызван немедленно), в более сложных приложениях очередь может содержать и другие события, чей приоритет выше, чем у нашего события `Paint`. В частности, если пользователь решит выйти из приложения, то это будет помечено как сообщение, известное под именем `WM_QUIT`.
- Если у вас есть более сложное, многопоточное приложение, вероятно, вы решите, чтобы только один его поток обрабатывал рисование. Применение `Invalidate()` для маршрутизации всех событий рисования через очередь сообщений обеспечивает хорошую возможность гарантировать, что один и тот же поток выполнит все рисование, независимо то того, какой поток его потребовал. (Это будет любой поток, отвечающий за очередь сообщений — тот, который запустил `Application.Run()`.)
- Существует еще одна причина, связанная с производительностью. Предположим, что есть два разных запроса на рисование части экрана, которые поступают почти одновременно. Может быть, код только что модифицировал документ и желает гарантировать его отображение, в то время как пользователь только что переместил другое окно, которое накрывало часть клиентской области нашего окна. За счет вызова `Invalidate()` мы предоставляем для Windows шанс узнать об этом. Затем Windows может объединить эти два события `Paint`, комбинируя недействительные области, чтобы собственно рисование произошло только один раз.

- Код, ответственный за перерисовку, вероятно, будет одним из сложнейших частей приложения, особенно если используется изощренный пользовательский интерфейс. Люди, которым придется сопровождать наш код в течение ряда лет, будут благодарны, если мы поместим весь код рисования в одном месте и сделаем его насколько возможно простым — иногда это легче сделать, если к нему ведет не слишком много путей из других частей программы.

В результате всего этого приходим к выводу, что лучше сосредоточить весь код рисования в процедуре `OnPaint()` либо в других методах, вызванных из данного. Однако необходимо сохранять разумный баланс: если вы захотите заменить единственный символ на экране, и вы точно знаете, что это не затронет ничего другого, что уже было нарисовано, то вы можете принять решение, что не стоит перегружать систему дополнительным `Invalidate()`, а лучше просто написать отдельную процедуру рисования.

В очень сложных приложениях вы можете даже написать целый класс, который отвечает за рисование экрана. Несколько лет назад, когда MFC была стандартной технологией разработки GDI-ориентированных приложений, библиотека MFC следовала именно такой модели — с применением класса C++ `C<ИмяПриложения>View`, отвечавшего за перерисовку. Однако даже в этом случае класс имел одну функцию-член `OnDraw()`, которая служила точкой входа для большинства запросов рисования.

Вычисление размеров элементов и размеров документа

В этом разделе мы вернемся к примеру `CapsEditor` и рассмотрим методы `CalculateLineWidths()` и `CalculateDocumentSize()`, вызываемые из `LoadFile()`.

```
private void CalculateLineWidths()
{
   Graphics dc = this.CreateGraphics();
   foreach (TextLineInformation nextLine in documentLines)
   {
       nextLine.Width = (uint)dc.MeasureString(nextLine.Text, mainFont).Width;
   }
}
```

Этот метод просто проходит по каждой прочитанной строке и использует метод `Graphics.MeasureString()` для вычисления того, сколько горизонтального пространства экрана ей требуется. Это значение сохраняется для последующего применения, потому что `MeasureString()` — дорогостоящая по вычислительным ресурсам операция. Если пример `CapsEditor` недостаточно прост, чтобы легко обработать высоту и местоположение каждого элемента, этот метод почти наверняка придется реализовать, чтобы также вычислить все эти величины.

Теперь, когда мы знаем размер каждого элемента на экране и можем вычислить местоположение каждого из них, у нас есть возможность получить реальный размер документа. Его высота определяется как количество строк, умноженное на высоту каждой строки. Ширина должна быть найдена в процессе итерации по всем строкам, чтобы найти самую длинную. Как высоту, так и ширину следует дополнить небольшими полями, чтобы документ выглядел более привлекательно.

Ниже приведен код метода, вычисляющего размер документа.

```
private void CalculateDocumentSize()
{
  if (!documentHasData)
  {
     documentSize = new Size(100, 200);
  }
```

```
    else
    {
       documentSize.Height = (int)(nLines*lineHeight) + 2*(int)margin;
       uint maxLineLength = 0;
       foreach (TextLineInformation nextWord in documentLines)
       {
         uint tempLineLength = nextWord.Width;
         if (tempLineLength > maxLineLength)
               maxLineLength = tempLineLength;
       }
       maxLineLength += 2*margin;
       documentSize.Width = (int)maxLineLength;
    }
    AutoScrollMinSize = documentSize;
  }
```

Сначала этот метод проверяет, есть ли данные, которые нужно отобразить. Если их нет, то мы прибегаем к хитрости и указываем жестко закодированный размер документа, который достаточен для отображения предупреждения <Empty document> (пустой документ) красного цвета. Если вы действительно хотите сделать это корректно, то нужно применить `MeasureString()`, чтобы определить действительный размер строки предупреждения.

Определив размер документа, мы сообщаем его экземпляру `Form`, устанавливая его свойство `Form.AutoScrollMinSize`. Когда мы это делаем, "за кулисами" происходит нечто интересное. В процессе установки значения этого свойства клиентская область объявляется недействительной и возбуждается событие `Paint` — по очень уважительной причине, связанной с тем, что изменение размера документа означает необходимость добавления или модификации линеек прокрутки, и вся клиентская область почти наверняка должна быть перерисована. Чем это интересно? Если вернуться к методу `LoadFile()`, то мы обнаружим, что вызов `Invalidate()` в этом методе на самом деле избыточен. Клиентская область в любом случае будет объявлена недействительной при установке размера документа. Явный вызов `Invalidate()` остается в `LoadFile()` для иллюстрации того, как следует поступать в общем случае. Но фактически в данном случае все повторные вызовы `Invalidate()` приводят только к избыточным запросам, дублирующим событие `Paint`. Однако это в свою очередь иллюстрирует то, как `Invalidate()` дает возможность Windows оптимизировать производительность. Второе событие `Paint` фактически не может быть возбуждено — Windows обнаружит, что в очереди уже есть событие `Paint`, и сравнит запрошенную недействительную область, чтобы проверить, нужно ли объединить ее с недействительной областью нового события. В данном случае оба события `Paint` объявляют недействительной всю клиентскую область окна, поэтому ничего не нужно делать, и Windows молча отбрасывает второй запрос `Paint`. Конечно, весь этот процесс потребует определенного процессорного времени, но это время совершенно незначительно по сравнению с тем, сколько требуется на перерисовку.

OnPaint()

Теперь, когда мы разобрались с тем, как `CapsEditor` загружает файл, самое время посмотреть, как будет выполняться его отображение:

```
protected override void OnPaint(PaintEventArgs e)
{
   base.OnPaint(e);
   Graphics dc = e.Graphics;
   int scrollPositionX = AutoScrollPosition.X;
   int scrollPositionY = AutoScrollPosition.Y;
   dc.TranslateTransform(scrollPositionX, scrollPositionY);
```

```
    if (!documentHasData)
    {
      dc.DrawString("<Empty document>", emptyDocumentFont,
      emptyDocumentBrush, new Point(20,20));
      base.OnPaint(e);
      return;
    }
    // поиск строк, находящихся в области отсечения
    int minLineInClipRegion = WorldYCoordinateToLineIndex(e.ClipRectangle.Top -
                              scrollPositionY);
    if (minLineInClipRegion == -1)
      minLineInClipRegion = 0;
    int maxLineInClipRegion =
    WorldYCoordinateToLineIndex(e.ClipRectangle.Bottom - scrollPositionY);
    if (maxLineInClipRegion >= documentLines.Count || maxLineInClipRegion == -1)
      maxLineInClipRegion = documentLines.Count-1;
    TextLineInformation nextLine;
    for (int i=minLineInClipRegion; i<=maxLineInClipRegion ; i++)
    {
      nextLine = (TextLineInformation)documentLines[i];
      dc.DrawString(nextLine.Text, mainFont, mainBrush,
      LineIndexToWorldCoordinates(i));
    }
  }
```

В сердце переопределенного метода `OnPaint()` находится цикл, проходящий по все строкам документа, вызывая `Graphics.DrawString()` для рисования каждой из них. Весь прочий код в основном предназначен для оптимизации рисования — обычное хозяйство, занимающееся поиском того, что именно нуждается в перерисовке — чтобы не заставлять `Graphics` тупо перерисовывать все подряд.

Все начинается с проверки того, есть ли вообще данные в документе. Если их нет, отображается сообщение об этом, вызывается метод `OnPaint()` базового класса и выполняется выход. Если данные есть, проверяется прямоугольник отсечения. Это делается вызовом другого метода — `WorldYCoordinateToLineIndex()`. Данный метод рассматривается следующим, но, по сути, он принимает позицию у относительно вершины документа и определяет, какая строка документа должна отображаться в этой точке.

При первом вызове метода `WorldYCoordinateToLineIndex()` ему передается значение координаты `(e.ClipRectangle.Top - scrollPositionY)`. Это — вершина области отсечения, преобразованная к мировым координатам. Если возвращаемое значение будет равно `-1`, то мы в безопасности и предполагаем, что нужно начинать с начала документа (это тот случай, когда области отсечения находится внутри верхнего поля отступа).

Один раз сделав это, мы по существу повторяем этот процесс до нижней границы прямоугольника отсечения, пока не найдем последнюю строку, находящуюся внутри него. Индексы первой и последней строк сохраняются, соответственно, в `minLineInClipRegion` и `maxLineInClipRegion`, так что остается только запустить цикл `for` между этими значениями, чтобы выполнить все рисование. Внутри цикла рисования нужно выполнить приблизительно преобразование, обратное тому, что реализовано в `WorldYCoordinateToLineIndex()`. Имеется индекс строки текста, и необходимо определить, где она должна быть отображена. Это вычисляется достаточно просто, но оно помещено в оболочку другого метода — `LineIndexToWorldCoordinates()`, — который возвращает требуемые координаты левого верхнего угла элемента. Возвращенные координаты относятся к системе мировых координат, но это нормально, потому что мы уже вызвали `TranslateTransform()` для объекта `Graphics`, поэтому ему нужно передать именно мировые координаты, а не страничные, когда запрашивается отображение элементов.

Преобразования координат

В этом разделе рассматривается реализация вспомогательных методов примера CapsEditor, служащих для преобразования координат. Речь идет о методах WorldYCoordinateToLineIndex() и LineIndexToWorldCoordinates(), которые упоминались в предыдущем разделе, наряду с парой других.

Первый метод, LineIndexToWorldCoordinates(), принимает индекс строки и вычисляет мировые координаты левого верхнего угла этой строки, используя известное значение отступа и высоту строки:

```
private Point LineIndexToWorldCoordinates(int index)
{
     Point TopLeftCorner = new Point(
            (int)margin, (int)(lineHeight*index + margin));
     return TopLeftCorner;
}
```

Мы также используем метод, который выполняет примерно обратную операцию, в OnPaint(). WorldYCoordinateToLineIndex() вычисляет индекс строки, но принимает во внимание только вертикальную мировую координату. Это потому, что она используется для определения индекса, соответствующего верху и низу области отсечения:

```
private int WorldYCoordinateToLineIndex(int y)
{
  if (y < margin)
  {
    return -1;
  }
  return (int)((y-margin)/lineHeight);
}
```

Есть еще три метода, которые будут вызваны из обработчика двойного щелчка кнопкой мыши пользователя. Во-первых, это метод, вычисляющий индекс строки, отображенной по заданным мировым координатам.

В отличие от WorldYCoordinateToLineIndex(), этот метод принимает обе координаты — x и y. Он возвращает -1, если по указанным координатам не обнаруживает строки:

```
private int WorldCoordinatesToLineIndex(Point position)
{
   if (!documentHasData)
   {
     return -1;
   }
   if (position.Y < margin || position.X < margin)
   {
     return -1;
   }
   int index = (int)(position.Y-margin)/(int)this.lineHeight;

   // если позиция за пределами документа
   if (index >= documentLines.Count)
   {
     return -1;
   }

   // проверить, что позиция по горизонтали находится в пределах строки
   TextLineInformation theLine =
                        (TextLineInformation)documentLines[index];
```

```
    if (position.X > margin + theLine.Width)
    {
      return -1;
    }

    // все в порядке, можно вернуть номер строки
    return index;
  }
```

И, наконец, при случае нам также понадобится выполнить преобразование между индексом строки и страничными, а не мировыми координатами. Сделать это позволят следующие методы:

```
private Point LineIndexToPageCoordinates(int index)
{
  return LineIndexToWorldCoordinates(index) +
  new Size(AutoScrollPosition);
}
private int PageCoordinatesToLineIndex(Point position)
{
  return WorldCoordinatesToLineIndex(position - new
  Size(AutoScrollPosition));
}
private Point LineIndexToPageCoordinates(int index)
{
  return LineIndexToWorldCoordinates(index) +
  new Size(AutoScrollPosition);
}
private int PageCoordinatesToLineIndex(Point position)
{
  return WorldCoordinatesToLineIndex(position - new
  Size(AutoScrollPosition));
}
```

Обратите внимание, что для преобразования *к* страничным координатам добавляется значение `AutoScrollPosition`, которое отрицательно.

Хотя эти методы сами по себе выглядят не особенно интересными, они иллюстрируют общую технику, которую, возможно, вам придется часто использовать. Работая с GDI+, вы часто попадаете в ситуацию, когда у вас есть определенные координаты (например, координаты пользовательского щелчка кнопкой мыши), и нужно найти, какой именно элемент расположен в этой точке. Или наоборот — имея определенный отображаемый элемент, нужно узнать, где он должен быть отображен. Поэтому, если вы пишете приложение GDI+, то всегда удобно иметь методы, выполняющие трансформации, подобные тем, что представлены здесь.

Реакция на пользовательский ввод

До сих пор все, что мы делали с этой главе с примером `CapsEditor`, за исключением меню File (Файл), делалось одним способом: приложение взаимодействовало с пользователем, отображая информацию на экране. Но почти все программы работают в двух направлениях: пользователь также может общаться с программой. Теперь мы добавим такую возможность в `CapsEditor`.

Заставить приложение GDI+ реагировать на ввод пользователя в действительности намного проще, чем написать код рисования экрана (в главе 31 описано, как обрабатывать пользовательский ввод). По сути, для этого необходимо переопределить методы из класса `Form`, которые вызываются из обработчиков событий, подобно тому, как вызывается `OnPaint()` при возникновении события `Paint`.

В табл. 33.5 перечислены методы, которые, возможно, потребуется переопределить, чтобы отреагировать на щелчки или перемещения мыши пользователем.

Таблица 33.5. Методы, работающие с мышью

Метод	Вызывается, когда…
`OnClick(EventArgs e)`	Выполнен щелчок кнопкой мыши.
`OnDoubleClick(EventArgs e)`	Выполнен двойной щелчок кнопкой мыши.
`OnMouseDown(MouseEventArgs e)`	Нажата левая кнопка мыши.
`OnMouseHover(MouseEventArgs e)`	Указатель мыши остановлен где-то после перемещения.
`OnMouseMove(MouseEventArgs e)`	Указатель мыши перемещен.
`OnMouseUp(MouseEventArgs e)`	Отпущена левая кнопка мыши.

Если нужно обнаружить, что пользователь вводит некоторый текст, возможно, потребуется переопределить методы, перечисленные в табл. 33.6.

Таблица 33.6. Методы, работающие с клавиатурой

Метод	Вызывается, когда…
`OnKeyDown(KeyEventArgs e)`	Клавиша нажата.
`OnKeyPress(KeyPressEventArgs e)`	Клавиша нажата и отпущена.
`OnKeyUp(KeyEventArgs e)`	Нажатая клавиша отпущена.

Обратите внимание, что некоторые из этих событий перекрываются. Например, если пользователь нажимает кнопку мыши, возбуждается событие `MouseDown`. Если кнопка немедленно отпущена, возбуждаются событие `MouseUp` и событие `Click`. К тому же некоторые из этих методов принимают аргумент, унаследованный от `EventArgs`, вместо экземпляра самого `EventArgs`. Экземпляры классов-наследников могут быть использованы для получения дополнительной информации об определенном событии. `MouseEventArgs` включает два свойства — `X` и `Y`, — которые сообщают координаты указателя мыши в момент нажатия ее кнопки. Как `KeyEventArgs`, так и `KeyPressEventArgs` имеют свойства, указывающие клавишу, к которой относится событие.

Об этом достаточно. Нам остается тщательно продумать логику, которую мы хотим реализовать. Единственное, что следует отметить — работая с приложениями GDI+, приходится реализовывать больше логики самостоятельно, чем в обычных приложениях Windows Forms. Это потому, что приложения Windows Forms обычно реагируют на более высокоуровневые события (например, `TextChanged` — для текстовых полей). В отличие от этого, события GDI+ более элементарны — пользователь щелкает кнопкой мыши или нажимает какую-то клавишу. Действия, предпринимаемые приложением, зависят, скорее, от последовательности событий, нежели от одного события. Например, если ваше приложение работает подобно Microsoft Word, где пользователь, чтобы выделить некоторый текст, нажимает левую кнопку мыши, затем перемещает мышь и отпускает кнопку мыши. Приложение принимает событие `MouseDown`, но с ним ничего нельзя сделать, кроме как запомнить, что кнопка была нажата с курсором в определенной позиции. Затем, когда принимается событие `MouseMove`, нужно проверить, где теперь находится курсор, и выделить текст, отмеченный пользователем, начиная с позиции нажатия кнопки. Когда пользователь отпускает левую кнопку мыши, то должно быть предпринято соответствующее действие (в методе `OnMouseUp()`),

чтобы проверить, не было ли выполнено перетаскивание, пока кнопка мыши была нажата, и соответствующим образом отреагировать. И только в этот момент последовательность завершается.

Другой момент, который необходимо отметить — это то, что поскольку некоторые события перекрываются, часто приходится делать выбор, на какое из них должен реагировать ваш код.

Золотое правило гласит, что нужно тщательно продумать логику каждой комбинации перемещения и щелчков мыши или событий клавиатуры, которые может инициировать пользователь, и гарантировать, чтобы ваше приложение реагировало так, чтобы обеспечивалось интуитивно ожидаемое поведение в *любой* ситуации. Большая часть работы здесь будет состоять в обдумывании, а не в кодировании, потому что нужно принимать во внимание большое число возможных комбинаций пользовательского ввода. Например, что должно делать приложение, если пользователь начнет вводить текст, пока нажата кнопка мыши? Это может показаться маловероятным, но рано или поздно какой-нибудь пользователь обязательно это попробует!

Пример `CapsEditor` ограничивается очень простыми случаями, поэтому нам не приходится обдумывать никакие комбинации подобного рода. Единственное, что нужно сделать — обеспечить реакцию на двойной щелчок мыши, и в этом случае перевести в верхний регистр все символы строки, на которой будет находиться в этот момент указатель мыши.

Это должно быть достаточно простой задачей, однако есть одна заминка. Необходимо перехватить событие `DoubleClick`, но приведенная выше табл. 33.5 сообщает, что это событие принимает параметр `EventArgs`, а не `MouseEventArgs`. Проблема в том, что нужно знать, где находится указатель мыши, когда пользователь выполняет двойной щелчок, чтобы правильно идентифицировать строку текста, которая должна быть переведена в верхний регистр — т.е. необходим параметр `MouseEventArgs`. Существует два обходных пути. Один заключается в использовании статического метода, реализованного объектом `Form1` — `Control.MousePosition` — чтобы найти позицию указателя мыши:

```
protected override void OnDoubleClick(EventArgs e)
{
   Point MouseLocation = Control.MousePosition;
   // обработать двойной щелчок
}
```

В большинстве случаев это будет работать. Однако может возникнуть проблема, связанная с тем, что наше приложение (или даже какое-нибудь другое приложение с высоким приоритетом) будет выполнять некоторую интенсивную вычислительную работу в тот момент, когда пользователь выполнит двойной щелчок. И тогда может случиться так, что `OnDoubleClick()` будет вызван через полсекунды или около того *после* реального двойного щелчка пользователя. Конечно, нам не нужны такие задержки, потому что они очень досаждают пользователям, а они иногда случаются по причинам, от нас не зависящим (на медленном компьютере, например). Полсекунды — достаточное время для того, чтобы указатель мыши был передвинут на пол-экрана, и в этом случае вызов `Control.MousePosition` вернет совершенно неверную позицию!

Лучше положиться на одно из многих перекрытий событий, связанных с мышью. Первая часть двойного щелчка — это нажатие левой кнопки мыши. Это значит, что если вызывается `OnDoubleClick()`, точно известно, что перед этим вызывается `OnMouseDown()` с указателем мыши, находящимся в той же позиции. Можно воспользоваться переопределенным методом `OnMouseDown()`, чтобы сохранить позицию мыши, подготовив ее для `OnDoubleClick()`. Именно такой подход мы используем в `CapsEditor`:

```
protected override void OnMouseDown(MouseEventArgs e)
{
   base.OnMouseDown(e);
   mouseDoubleClickPosition = new Point(e.X, e.Y);
}
```

Теперь взглянем на переопределение `OnDoubleClick()`. Здесь работы побольше:

```
protected override void OnDoubleClick(EventArgs e)
{
   int i = PageCoordinatesToLineIndex(mouseDoubleClickPosition);
   if (i >= 0)
   {
     TextLineInformation lineToBeChanged =
                          (TextLineInformation)documentLines[i];
     lineToBeChanged.Text = lineToBeChanged.Text.ToUpper();
     Graphics dc = CreateGraphics();
     uint newWidth =(uint)dc.MeasureString(lineToBeChanged.Text,
                                           mainFont).Width;
     if (newWidth > lineToBeChanged.Width)
           lineToBeChanged.Width = newWidth;
     if (newWidth+2*margin > this.documentSize.Width)
     {
           documentSize.Width = (int)newWidth;
           AutoScrollMinSize = this.documentSize;
     }
     Rectangle changedRectangle = new Rectangle(
                                        LineIndexToPageCoordinates(i),
                                        new Size((int)newWidth,
                                        (int)this.lineHeight));
     this.Invalidate(changedRectangle);
   }
   base.OnDoubleClick(e);
}
```

Здесь мы начинаем с вызова `PageCoordinatesToLineIndex()`, чтобы определить, на какой строке текста находился указатель мыши в момент двойного щелчка. Если этот вызов вернет `-1`, значит, указатель был вне пределов текста, и потому ничего делать не нужно, за исключением, конечно, вызова `OnDoubleClick()` базового класса, чтобы позволить Windows выполнить обработку по умолчанию.

Предположим, мы идентифицировали строку текста. Тогда можно применить метод `string.ToUpper()`, чтобы преобразовать ее в верхний регистр. Это — простая часть работы. Сложная часть заключается в том, чтобы определить, что и где должно быть перерисовано. К счастью, поскольку этот пример прост, комбинаций не так много. Можно предположить, что преобразование строки в верхний регистр всегда либо оставит длину строки неизменной, либо увеличит ее. Заглавные буквы крупнее прописных, а потому ширина никогда не станет меньше. Также нам известно, что поскольку перенос строк не выполняется, наша строка текста не перейдет на следующую и не вытолкнет остальные вниз. Значит, преобразование в верхний регистр не изменит местоположения никаких других элементов. Это значительно упрощает задачу!

Следующее, что делает наш код — он использует `Graphics.MeasureString()` для того, чтобы вычислить новую ширину текста. И здесь возникают две описанных ниже возможности.

- Новая ширина строки может сделать ее самой длинной в тексте, что увеличит ширину всего документа. В этом случае придется установить `AutoScrollMinSize` новое значение, чтобы правильно работали линейки прокрутки.
- Размер документа может остаться прежним.

В любом случае потребуется перерисовать экран, вызвав `Invalidate()`. Но изменилась только одна строка, а потому нам не нужно перерисовывать весь документ. Вместо этого необходимо найти границы прямоугольной области, содержащей модифицированную строку, чтобы их можно было передать `Invalidate()`, обеспечивая перерисовку только измененной строки. И это именно то, что делает показанный ранее код. Наш вызов `Invalidate()` инициирует вызов `OnPaint()`, когда обработчик события мыши возвращает управление. Памятуя о ранее приведенных предупреждениях относительно сложности установки отладочных точек прерывания внутри `OnPaint()`, мы запускаем пример и устанавливаем точку прерывания в `OnPaint()`, чтобы перехватить результирующее действие перерисовки, в результате чего обнаруживаем, что параметр `PaintEventArgs` метода `OnPaint()` действительно содержит область отсечения, соответствующую указанному прямоугольнику с измененной строкой. И поскольку мы переопределили `OnPaint()` так, что он тщательно принимает во внимание область отсечения, подлежащую перерисовке, то как раз одна измененная строка и будет перерисована.

Печать

До сих пор в этой главе мы сосредоточили свое внимание исключительно на рисовании на экране. Тем не менее, однажды вам понадобится получить твердую копию данных. Это и есть тема настоящего раздела. Мы расширим пример `CapsEditor`, добавив возможность предварительно просмотра копии для печати и собственно печати редактируемого документа.

К сожалению, у нас нет достаточного места, чтобы рассмотреть тему печати очень подробно, поэтому мы реализуем в примере лишь самую базовую функциональность печати. Обычно, реализуя возможность печати данных в приложении, необходимо добавить три команды в главное меню приложения File (Файл):

- Page Setup (Параметры страницы) — для выбора опций печати, например, какие страницы печатать, какой принтер использовать и тому подобное;
- Print Preview (Предварительный просмотр) — для открытия новой формы, демонстрирующей внешний вид макета для печати;
- Print (Печать) — для отправки документа на устройство печати.

Пока для простоты мы не будем реализовывать команду меню Page Setup. Печать будет выполняться только с установками по умолчанию. Однако отметим, что если вы пожелаете реализовать Page Setup, то для вас Microsoft подготовила класс диалогового окна настройки печати — `System.Windows.Forms.PrintDialog`. Вы можете написать обработчик события меню, который отобразит эту форму и сохранит выбранные пользователем настройки.

Во многих отношениях печать — это то же самое, что отображение на экране. Вы используете контекст устройства (экземпляр `Graphics`) и вызываете обычные команды отображения с этим экземпляром. В Microsoft разработали множество классов, призванных помочь в этом. Вот два главных класса, которые вам для этого понадобятся:

- `System.Drawing.Printing.PrintDocument`
- `System.Drawing.Printing.PrintPreviewDialog`

Эти два класса берут на себя обеспечение доставки команд рисования нужному контексту устройств для выполнения печати, избавляя разработчика от рутины и позволяя сосредоточиться на логике того, что необходимо напечатать.

Несмотря на эти сложности, сам процесс печати довольно прост. Необходимые шаги программного характера, которые для этого нужно выполнить, сводятся к описанным ниже.

- **Печать**. Вы создаете экземпляр объекта `PrintDocument` и вызываете его метод `Print()`. Этот метод посылает событие `PrintPage` для печати первой страницы. `PrintPage` принимает параметр `PrintPageEventArgs`, который содержит информацию относительно размера бумаги и настроек, а также объект `Graphics`, используемый для команд рисования. Таким образом, вам нужно написать обработчик для этого события и реализовать его для печати страницы. Этот обработчик также должен устанавливать свойство `PrintPageEventArgs` булевского типа под названием `HasMorePages` в `true` или `false`, для указания того, есть ли еще страницы для печати. Метод `PrintDocument.Print()` выполняет повторяющуюся инициализацию события `PrintPage` до тех пор, пока не увидит, что `HasMorePages` установлено в `false`.
- **Предварительный просмотр**. В этом случае создаются экземпляры объектов `PrintDocument` и `PrintPreviewDialog`. Экземпляр `PrintDocument` присоединяется к `PrintPreviewDialog` (через свойство `PrintPreviewDialog.Document`), после чего вызывается метод диалога `ShowDialog()`. Этот метод модально отображает диалоговое окно, которое представляет собой стандартную форму Windows для предварительного просмотра печати с загруженными в нее страницами документа. Внутренне страницы отображаются последовательной генерацией события `PrintPage` до тех пор, пока свойство `HasMorePages` не станет равно `false`. При этом нет необходимости писать отдельный обработчик событий для этого; используется тот же обработчик, что служит для печати каждой страницы, поскольку код рисования в обоих случаях идентичен (в конце концов, предварительный просмотр печати должен показывать в точности то, что будет напечатано).

Реализация команд меню Print и Print Preview

Теперь, когда мы в общих чертах описали этот процесс, в данном разделе посмотрим, как это выглядит в исходном коде. Исходный код примера доступен в виде проекта `PrintingCapsEdit` на прилагаемом компакт-диске. Он состоит из проекта `CapsEditor` с дополнениями, выделенными в приведенном ниже фрагменте.

Начнем с использования представления конструктора интегрированной среды разработки Visual Studio 2008, добавив две новых команды в меню File (Файл), а именно: Print (Печать) и Print Preview (Предварительный просмотр). С помощью окна свойств переименуем их в `menuFilePrint` и `menuFilePrintPreview`, а также сделаем их неактивными при запуске приложения (нельзя ничего печатать, пока документ не открыт). Эти команды меню сделаем активными, добавив следующий код в метод `LoadFile()` главной формы, который отвечает за загрузку файла в приложение `CapsEditor`:

```
private void LoadFile(string FileName)
{
  StreamReader sr = new StreamReader(FileName);
  string nextLine;
  documentLines.Clear();
  nLines = 0;
  TextLineInformation nextLineInfo;
  while ((nextLine = sr.ReadLine()) != null)
  {
     nextLineInfo = new TextLineInformation();
     nextLineInfo.Text = nextLine;
```

```
      documentLines.Add(nextLineInfo);
      ++nLines;
    }
    sr.Close();
    if (nLines > 0)
    {
      documentHasData = true;
      menuFilePrint.Enabled = true;
      menuFilePrintPreview.Enabled = true;
    }
    else
    {
      documentHasData = false;
      menuFilePrint.Enabled = false;
      menuFilePrintPreview.Enabled = false;
    }
    CalculateLineWidths();
    CalculateDocumentSize();
    Text = standardTitle + " - " + FileName;
    Invalidate();
  }
```

Здесь выделен новый код, добавленный к методу. Далее добавим поле-член в класс `Form1`:

```
  public partial class Form1 : Form
  {
    private int pagesPrinted = 0;
```

Это поле будет использоваться для указания номера текущей печатаемой страницы. Мы объявляем его полем-членом, потому что информацию о текущей странице нужно держать в памяти между вызовами обработчика событий `PrintPage`.

Далее следуют обработчики событий команд меню `Print` и `PrintPreview`:

```
  private void menuFilePrintPreview_Click(object sender, System.EventArgs e)
  {
    this.pagesPrinted = 0;
    PrintPreviewDialog ppd = new PrintPreviewDialog();
    PrintDocument pd = new PrintDocument();
    pd.PrintPage += new PrintPageEventHandler
       (this.pd_PrintPage);
    ppd.Document = pd;
    ppd.ShowDialog();
  }
  private void menuFilePrint_Click(object sender, System.EventArgs e)
  {
    this.pagesPrinted = 0;
    PrintDocument pd = new PrintDocument();
    pd.PrintPage += this.pd_PrintPage);
    pd.Print();
  }
```

Мы уже описали шаги, которые необходимо выполнить для осуществления печати, и можно видеть, что эти обработчики событий просто реализуют описанную процедуру. В обоих случаях создается экземпляр объекта `PrintDocument` и к его событию `PrintPage` присоединяется обработчик. В случае печати вызывается `PrintDocument.Print()`, в то время как для предварительного просмотра объект `PrintDocument` присоединяется к `PrintPreviewDialog`, и вызывается метод `ShowDialog()` диалогового окна предварительного просмотра. Реальная работа события `PrintPage` выполняется в обработчике события, который выглядит следующим образом:

```
private void pd_PrintPage(object sender, PrintPageEventArgs e)
{
  float yPos = 0;
  float leftMargin = e.MarginBounds.Left;
  float topMargin = e.MarginBounds.Top;
  string line = null;
  // Вычислить количество строк на страницу.
  int linesPerPage = (int)(e.MarginBounds.Height / mainFont.GetHeight(e.Graphics));
  int lineNo = pagesPrinted * linesPerPage;
  // Печатать каждую строку файла.
  int count = 0;
  while(count < linesPerPage && lineNo < this.nLines)
  {
     line = ((TextLineInformation)this.documentLines[lineNo]).Text;
     yPos = topMargin + (count * mainFont.GetHeight(e.Graphics));
     e.Graphics.DrawString(line, mainFont, Brushes.Blue,
       leftMargin, yPos, new StringFormat());
     lineNo++;
     count++;
  }
  // Если еще остались строки, печатать другую страницу.
  if(this.nLines > lineNo)
     e.HasMorePages = true;
  else
     e.HasMorePages = false;
  pagesPrinted++;
}
```

После объявления нескольких локальных переменных первое, что мы здесь делаем — это вычисляем количество строк, которое может быть отображено на одной странице, как высоту страницы, деленную на высоту строки с округлением в меньшую сторону.

Высоту страницы можно получить из свойства `PrintPageEventArgs.MarginBounds`. Это свойство представляет собой структуру `RectangleF`, которая инициализирована размерами страницы. Высота строки получается из поля `Form1.mainFont`, представляющего шрифт, используемый для отображения текста. Нет причин использовать для печати какой-то другой шрифт.

Обратите внимание, что в примере `PrintingCapsEditor` количество строк на страницу всегда одно и то же, поэтому имело бы смысл его запомнить после первого вычисления. Однако эти вычисления не особенно сложны, к тому же в более развитом приложении это значение может меняться, поэтому ничего плохого не случится, если мы станем повторно вычислять его при печати каждой страницы.

Мы также инициализируем здесь переменную с именем `lineNo`. Она представляет начинающийся с нуля индекс строки документа, которая будет первой на странице. Эта информация важна, поскольку в принципе метод `pd_PrintPage()` может быть вызван для печати любой страницы, а не только первой. Значение `lineNo` вычисляется как количество строк на странице, умноженное на количество уже напечатанных страниц.

Далее запускается цикл, печатающий каждую строку. Этот цикл прерывается либо когда обнаруживается, что уже напечатаны все строки текста документа, либо когда будут напечатаны все строки, помещающиеся на текущей страницы — в зависимости от того, что произойдет раньше. И, наконец, мы проверяем, остались ли еще строки документа для печати, и соответствующим образом устанавливаем свойство `HasMorePages` аргумента `PrintPageEventArgs`, а также увеличиваем на единицу значение поля `pagesPrinted`, чтобы знать, какую страницу следует напечатать, когда в следующий раз будет вызван обработчик события `PrintPage`.

Следует отметить один момент относительно этого обработчика событий, а именно: нам не нужно беспокоиться о том, как выполняются команды рисования. Мы просто используем объект `Graphics`, который получаем из `PrintPageEventArgs`. Класс `PrintDocument`, разработанный Microsoft, сам позаботится о том, что если мы собираемся печатать, то объект `Graphics` должен быть "прикреплен" к принтеру. Если же речь идет о предварительном просмотре, он "прикрепляется" к форме предварительного просмотра на экране.

И, наконец, мы должны убедиться, что пространство имен `System.Drawing.Printing` включено для поиска определений типов:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Drawing.Printing;
using System.Text;
using System.Windows.Forms;
using System.IO;
```

Все, что остается — скомпилировать проект и проверить, как он работает. На рис. 33.19 показано то, что мы увидим, если запустим `CapsEdit`, загрузим текстовый документ (как и раньше — файл исходного кода на C# из проекта) и выберем команду меню Print Preview (Предварительный просмотр).

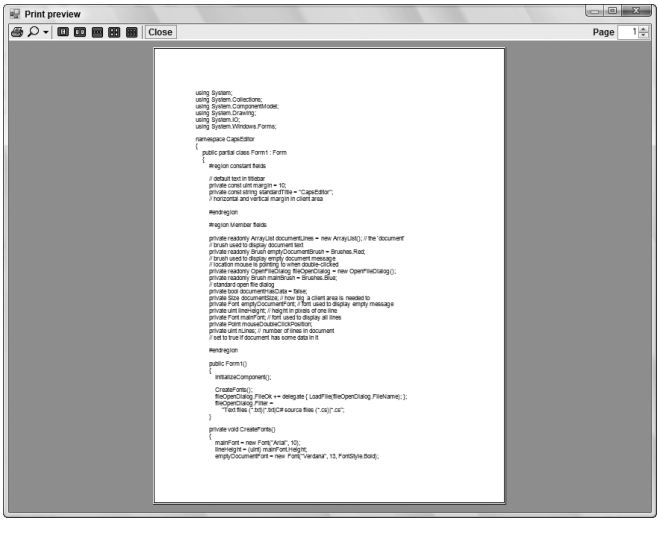

Рис. 33.19. Результат работы примера `CapsEdit`

На рис. 33.19 показан документ, прокрученный до 5-й страницы, и установлен нормальный размер для предварительного просмотра. `PrintPreviewDialog` предлагает множество средств, что видно в панели инструментов в верхней части формы. Доступные опции включают печать документа, увеличение его, отображение двух, трех, четырех или шести страниц сразу. Все эти опции полностью функциональны, и нам не приходится ничего делать. На рис. 33.20 можно видеть результат переключения масштаба на автоматический режим и выбора одновременного отображения четырех страниц (третья справа кнопка в линейке инструментов).

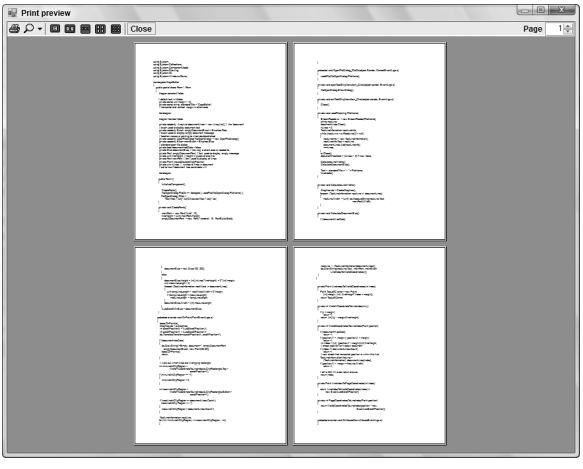

Рис. 33.20. Результат работы примера `CapsEdit` при изменении опций отображения

Резюме

Эта глава была посвящена рисованию на устройстве отображения, когда оно выполняется вашим собственным кодом, а не за счет использования предопределенных элементов управления и диалоговых окон — т.е. с применением GDI+. GDI+ — это мощный инструмент, включающий большое количество базовых классов .NET, призванных помочь разработчику выполнять рисование на устройстве. Мы увидели, что процесс рисования на самом деле сравнительно прост — в большинстве случаев, чтобы нарисовать текст или сложные фигуры либо же отобразить графический образ, достаточно всего пары операторов на C#. Однако управление рисованием — это сложная работа, выполняемая "за кулисами"; она включает вычисление того, что нужно нарисовать, где именно, при каких условиях необходима перерисовка, а при каких нет — все это более сложно и требует тщательного проектирования алгоритмов. По этой причине так же важно хорошо понимать принципы работы GDI+ и то, какие именно действия предпринимает Windows, когда требуется что-то нарисовать. В частности, из-за особенностей архитектуры Windows важно, где только возможно, инициировать рисование путем объявления областей экрана недействительными, чтобы Windows в нужное время посылала события `Paint`.

В .NET существует намного больше классов, используемых для рисования, чем мы смогли описать в настоящей главе. Однако если вы внимательно прочитали эту главу и поняли принципы рисования, то легко сможете расширить свои знания, изучая списки классов и методов в документации SDK и экспериментируя с ними. В конце отметим, что рисование, как и почти все другие аспекты программирования, требует логического, тщательного продумывания, четких алгоритмов, если вы хотите выйти за пределы применения стандартных элементов управления. Благодаря этому, разрабатываемое вами программное обеспечение значительно выиграет в части дружественности к пользователю и визуальной привлекательности. Многие приложения в организации пользовательского интерфейса полностью полагаются на стандартные элементы управления. И хотя это может оказаться эффективным, такие приложения получаются очень похожими друг на друга. Добавив некоторый код GDI+, обеспечивающий специфическое отображение, вы можете придать своим программам некоторую оригинальность и исключительность, что поможет увеличить их продажи!

В следующей главе мы поговорим о самой последней технологии презентаций на рабочих станциях — Windows Presentation Foundation (WPF).