ДОПОЛНИТЕЛЬНАЯ ГЛАВА 3

Визуализация графических данных с помощью GDI+

В предыдущей главе вы ознакомились с процессом построения настольных приложений с графическим интерфейсом пользователя на основе `System.Windows.Forms`. Целью этой главы будет изучение деталей визуализации графики (включая стилизованный текст и графические данные) на поверхности `Form`. Мы начнем с общего обзора многочисленных пространств имен, связанных с рисованием, рассмотрим роль события `Paint` и могущественного объекта `Graphics`.

В оставшейся части главы рассматриваются способы манипуляций цветами, шрифтами, геометрическими фигурами и графическими изображениями. Мы рассмотрим также ряд приемов визуализации графики, таких как непрямоугольная проверка попаданий, логика перетаскивания и формат ресурсов .NET. Хотя технически ресурсы не являются частью GDI+, часто они включают в себя манипуляции графическими данными (которые довольно-таки тесно связаны с темой GDI+).

На заметку! Если вы — веб-программист по призванию, то можете подумать, что GDI+ предназначено не для вас. Однако применение GDI+ не ограничивается традиционными настольными приложениями, и не менее существенно для веб-приложений.

Обзор пространств имен GDI+

Платформа .NET предоставляет ряд пространств имен, посвященных визуализации двухмерной графики. В дополнение к базовой функциональности вы найдете также набор инструментов графики (цвета, шрифты, перья, кисти и т.п.), а также типы, выполняющие геометрические трансформации, сглаживание, смешивание палитр и поддержку печати документов. Все вместе эти пространства имен образуют средство .NET под названием GDI+, которое представляет собой управляемую альтернативу программному интерфейсу Win32 Graphical Device Interface (GDI). В табл. 3.1 приведен общий обзор основных пространств имен GDI+.

Таблица 3.1. Основные пространства имен GDI+

Пространство имен	Назначение
`System.Drawing`	Это центральное пространство имен GDI+, определяющее многочисленные типы для базовой визуализации (шрифты, перья, базовые кисти и т.п.), а также мощный тип `Graphics`.
`System.Drawing.Drawing2D`	Это пространство имен представляет типы, используемые для более развитой функциональности двухмерной векторной графики (например, градиентные кисти, концы перьев, геометрические трансформации и т.п.).
`System.Drawing.Imaging`	Это пространство имен определяет типы, позволяющие манипулировать графическими изображениями (изменять палитру, извлекать метаданные изображения, манипулировать метафайлами и т.п.).
`System.Drawing.Printing`	Это пространство имен определяет типы, позволяющие визуализировать изображения для печатных страниц, взаимодействовать с принтером, форматировать общее представление конкретного задания печати.
`System.Drawing.Text`	Это пространство имен позволяет манипулировать коллекциями шрифтов.

На заметку! Все пространства имен GDI+ определены в сборке `System.Drawing.dll`. Хотя многие типы проектов Visual Studio автоматически устанавливают ссылки на эту библиотеку кода, при необходимости можно манипулировать ссылкой на `System.Drawing.dll`, используя диалоговое окно Add References (Добавление ссылок).

Обзор пространства имен `System.Drawing`

Подавляющее большинство типов, которые вы будете использовать при программировании приложений GDI+, находятся внутри пространства имен `System.Drawing`. Как и можно было ожидать, здесь есть классы, представляющие изображения, кисти, перья и шрифты. Более того, `System.Drawing` определяет множество связанных служебных типов, таких как `Color`.`Point` и `Rectangle`. В табл. 3.2 перечислены некоторые основные типы.

Таблица 3.2. Основные типы пространства имен `System.Drawing`

Тип	Назначение
`Bitmap`	Тип инкапсулирует данные изображения (`*.bmp` и т.п.).

`Brush`
`Brushes`
`SolidBrush`
`SystemBrushes`
`TextureBrush`
	Объекты кистей используются для заполнения внутренних поверхностей графических фигур, таких как прямоугольники, эллипсы и многоугольники.

`BufferedGraphics`	Этот тип представляет собой графический буфер для двойной буферизации, который используется для сокращения мерцания, вызванного перерисовкой поверхности отображения.

Окончание табл. 3.2

Тип	Назначение
`Color` `SystemColors`	Типы `Color` и `SystemColors` определяют множество статических, доступных только для чтения свойств, используемых для получения определенных цветов для конструирования различных перьев и кистей.
`Font` `FontFamily`	Тип `Font` инкапсулирует характеристики шрифта (название, жирность, курсив, размер и т.п.). `FontFamily` предоставляет абстракцию для группы шрифтов, имеющих сходный дизайн, но различия в стиле.
`Graphics`	Этот центральный класс представляет действительную поверхность рисования, а также множество методов для визуализации текста, изображений и геометрических шаблонов.
`Icon` `SystemIcons`	Эти классы представляют специальные пиктограммы, а также набор стандартных системных пиктограмм.
`Image` `ImageAnimator`	`Image` — абстрактный базовый класс, предоставляющий функциональность для типов `Bitmap`, `Icon` и `Cursor`. Класс `ImageAnimator` предлагает способ итерации по множеству производных от `Image` типов в течение заданного интервала времени.

`Pen`
`Pens`
`SystemPens`
	Перья — это объекты, используемые для рисования прямых и кривых линий. Тип `Pen` определяет ряд статических свойств, возвращающих `Pen` заданного цвета.

`Point` `PointF`	Эти структуры представляют координаты (*x*, *y*), отображающиеся на целые числа или числа с плавающей точкой, соответственно.
`Rectangle` `RectangleF`	Эти структуры представляют прямоугольные области (отображающиеся на целые числа или числа с плавающей точкой).
`Size` `SizeF`	Эти структуры представляют ширину/высоту (отображающиеся на целые числа или числа с плавающей точкой).
`StringFormat`	Этот тип используется для инкапсуляции различных средств текстовой компоновки (например, выравнивание, межстрочный интервал и т.п.).
`Region`	Этот тип описывает внутреннюю часть геометрического образа, состоящего из прямоугольников и путей.

Служебные типы `System.Drawing`

Многие из методов рисования, определенных в объекте `System.Drawing.Graphics`, требуют спецификации позиции или области, в которой вы хотите визуализировать данный элемент. Например, метод `DrawString()` требует указания места для визуализации текстовой строки на поверхности объекта типа-наследника `Control`. Учитывая, что `DrawString()` многократно перегружен, этот позиционный параметр может быть специфицирован с использованием координат (*x*, *y*) или границ "рамки", в пределах которой нужно рисовать. Другие методы типа GDI+ могут потребовать указания ширины и высоты заданного элемента либо внутренних границ геометрического изображения.

Чтобы специфицировать такую информацию, в пространстве имен `System.Drawing` определены типы `Point`, `Rectangle`, `Region` и `Size`. Очевидно, что `Point` представляет точку с координатами (`x`, `y`). Тип `Rectangle` охватывает пару точек, представляющих верхний левый и нижний правый границы прямоугольной области. Тип `Size` подобен `Rectangle`, но эта структура представляет размеры, заданные длиной и шириной.

И, наконец, Region обеспечивает способ представления и квалификации непрямоугольных поверхностей.

Переменные-члены, используемые типами Point, Rectangle и Size, внутренне представлены целочисленным типом данных. Если необходим более высокий уровень точности, можете использовать соответствующие типы PointF, RectangleF и SizeF, которые (как несложно догадаться), используют координаты с плавающей точкой. Но независимо от внутреннего представления данных, каждый тип имеет идентичный набор членов, включая ряд перегруженных операций.

Тип Point (PointF)

Первый служебный тип, о котором вам следует знать — это System.Drawing.Point (PointF). В отличие от иллюстративных типов Point, которые создавались в начальных главах, тип GDI+ Point (PointF) поддерживает ряд полезных членов, включая:

- +, -, ==, != — тип Point перегружает различные операции C#;
- X, Y — эти члены предоставляют доступ к лежащим в основе Point значениям (*x*, *y*);
- IsEmpty — этот член возвращает true, если *x* и *y* равны 0.

Для иллюстрации работы со служебными типами GDI+ рассмотрим пример консольного приложения (по имени UtilTupes), которое использует тип System.Drawing.Point (не забудьте ссылку на System.Drawing.dll):

```
using System;
using System.Drawing;
namespace UtilTypes
{
  public class Program
  {
    static void Main(string[] args)
    {
      // Создание и смещение точки.
      Point pt = new Point(100, 72);
      Console.WriteLine(pt);
      pt.Offset(20, 20);
      Console.WriteLine(pt);
      // Перегруженные операции Point.
      Point pt2 = pt;
      if(pt == pt2)
        WriteLine("Points are the same");
      else
        WriteLine("Different points");
      // Изменить в pt2 значение X.
      pt2.X = 4000;
      // Теперь показать каждое значение X.
      Console.WriteLine("First point: {0} ", pt);
      Console.WriteLine("Second point: {0} ", pt2);
      Console.ReadLine();
    }
  }
}
```

Тип Rectangle (RectangleF)

Тип Rectangle, как и Point, полезен во многих приложениях (как на основе GDI, так и нет). Один из наиболее полезных методов типа Rectangle — это Contains(). Этот метод позволяет определять, находится ли данный экземпляр Point или Rectangle в

пределах текущих границ другого объекта. Далее в этой главе вы увидите применение этого метода для выполнения проверки попадания на изображения GDI+. А пока рассмотрим простой пример:

```
static void Main(string[] args)
{
  ...
  // Point изначально вне границ прямоугольника.
  Rectangle r1 = new Rectangle(0, 0, 100, 100);
  Point pt3 = new Point(101, 101);
  if(r1.Contains(pt3))
    Console.WriteLine("Point is within the rect!");
  else
    Console.WriteLine("Point is not within the rect!");

  // Теперь поместим точку внутрь прямоугольника.
  pt3.X = 50;
  pt3.Y = 30;
  if(r1.Contains(pt3))
    Console.WriteLine("Point is within the rect!");      // внутри
  else
    Console.WriteLine("Point is not within the rect!");  // не внутри
  Console.ReadLine();
}
```

Класс Region

Тип `Region` представляет внутреннюю часть геометрической фигуры. Исходя из этого определения, имеет смысл, что конструкторы класса `Region` требуют передачи им некоторого существующего геометрического шаблона. Например, предположим, что создан прямоугольник размером 100×100 пикселей. Если вы хотите получить доступ к внутренней области этого прямоугольника, то можете написать нечто вроде такого:

```
// Получить внутреннюю часть прямоугольника.
Rectangle r = new Rectangle(0, 0, 100, 100);
Region rgn = new Region(r);
```

Имея внутренние размеры заданной фигуры, можно манипулировать ею, используя различные члены этого класса, включая перечисленные ниже.

- `Complement()`. Обновляет `Region` частью специфицированного графического объекта, который не пересекается с данным `Region`.
- `Exclude()`. Обновляет этот `Region` во внутренней части, которая не пересекается с указанным графическим объектом.
- `GetBounds()`. Возвращает `Rectangle(F)`, представляющий прямоугольную область, охватывающую данный `Region`.
- `Intersect()`. Обновляет этот `Region` пересечением его самого с указанным объектом графики.
- `Transform()`. Трансформирует `Region` указанным объектом `Matrix`.
- `Union()`. Обновляет этот `Region` объединением его самого с указанным объектом графики.
- `Translate()`. Смещает координаты данного `Region` на заданную величину.

Наверняка вы поняли основную идею, положенную в основу этих координатных примитивов. Если нужны дополнительные детали, обращайтесь к документации по .NET Framework 2.0 SDK.

На заметку! Типы `Size` и `SizeF` требуют лишь небольшого комментария. Каждый из этих типов определяет свойства `Height` и `Width`, а также несколько перегруженных операций.

Исходный код. Проект `UtilTypes` включен в подкаталог `Bonus Chapter 3`.

Класс `Graphics`

Класс `System.Drawing.Graphics` — это ворота в функциональность визуализации GDI+. Этот класс не только представляет поверхность рисования (такую, как поверхность формы, поверхность элемента управления или область в памяти), но также определяет десятки членов, которые позволяют визуализировать текст, изображения (пиктограммы, битовые карты и т.п.), и многочисленные геометрические шаблоны. В табл. 3.3 приведен неполный список членов этого класса.

Таблица 3.3. Члены класса `Graphics`

Метод	Назначение

`FromHdc()`
`FromHwnd()`
`FromImage()`
	Эти статические методы предоставляют возможность получения корректного объекта `Graphics` от заданного изображения (пиктограммы, битовой карты и т.п.) или виджета GUI.

`Clear()`	Этот метод заполняет объект `Graphics` указанным цветом, стирая в процессе текущую поверхность рисования.

`DrawArc()`
`DrawBeziers()`
`DrawCurve()`
`DrawEllipse()`
`DrawIcon()`
`DrawLine()`
`DrawLines()`
`DrawPie()`
`DrawPath()`
`DrawRectangle()`
`DrawRectangles()`
`DrawString()`
	Эти методы используются для визуализации заданного образа или геометрического шаблона. Все методы `Draw`*`XXX`*`()` требуют использования объектов GDI+ `Pen`.

`FillEllipse()`
`FillPie()`
`FillPolygon()`
`FillRectangle()`
`FillPath()`
	Эти методы используются для заполнения внутренности заданной геометрической фигуры. Все методы `Fill`*`XXX`*`()` требуют применения объектов GDI+ `Brush`.

Вдобавок к предоставлению множества методов для визуализации, класс `Graphics` определяет дополнительные члены, позволяющие конфигурировать "состояние" объекта `Graphics`. Присваивая значения свойствам, перечисленным в табл. 3.4, можно изменять текущие операции визуализации.

На заметку! Что касается ASP.NET, то `System.Drawing` предоставляет тип `BufferedGraphics`, который позволяет визуализировать графику с использованием системы двойной буферизации для минимизации или исключения мерцания, которое может возникнуть во время операций визуализации. Подробности ищите в документации по .NET Framework 2.0 SDK.

Таблица 3.4. Свойства состояния класса `Graphics`

Свойства	Назначение

`Clip`
`ClipBounds`
`VisibleClipBounds`
`IsClipEmpty`
`IsVisibleClipEmpty`
	Эти свойства позволяют устанавливать опции отсечения, используемые с текущим объектом `Graphics`.

`Transform`	Это свойство позволяет трансформировать "мировые координаты" (подробнее об этом речь пойдет ниже).

`PageUnit`
`PageScale`
`DpiX`
`DpiY`
	Эти свойства позволяют конфигурировать точку начала координат операций визуализации, а также единицу измерения.

`SmoothingMode`
`PixelOffsetMode`
`TextRenderingHint`
	Эти свойства позволяют конфигурировать гладкость геометрических объектов и текста.

`CompositingMode` `CompositingQuality`	Свойство `CompositingMode` определяет, будет ли рисование перекрывать фон или смешиваться с ним.
`InterpolationMode`	Это свойство специфицирует интерполяцию данных между конечными точками.

Вопреки предположениям, класс `Graphics` невозможно создать через ключевое слово `new`, поскольку в нем нет общедоступно объявленных конструкторов. Как же тогда получить действительный объект `Graphics`? Давайте посмотрим.

Сеансы `Paint`

Наиболее распространенный способ получения объекта `Graphics` состоит в обработке события `Paint`. Вспомните из предыдущей главы, что класс `Control` определяет виртуальный метод по имени `OnPaint()`. Когда вы хотите, чтобы `Form` визуализировал графические данные на своей поверхности, можете переопределить этот метод и извлечь объект `Graphics` из входного параметра `PaintEventArgs`. Для иллюстрации создайте новое приложение Windows Forms по имени `BasicPaintForm` и обновите класс-наследник `Form` следующим образом:

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
    CenterToScreen();
    this.Text = "Basic Paint Form";
  }
  protected override void OnPaint(PaintEventArgs e)
  {
    // Переопределяя OnPaint(), не забудьте вызвать реализацию базового класса.
    base.OnPaint(e);
    // Получить объект Graphics из входящего PaintEventArgs.
    Graphics g = e.Graphics;
    // Визуализировать текстовое сообщение в заданном шрифте и цвете.
    g.DrawString("Hello GDI+", new Font("Times New Roman", 20), Brushes.Green, 0, 0);
  }
}
```

Хотя переопределение `OnPaint()` допустимо, более распространенный способ обработки события `Paint` предусматривает использование ассоциированного делегата `PaintEventHandler` (фактически это поведение по умолчанию, принятое в Visual Studio при обработке событий через окно Properties). Этот делегат может указывать на любой метод, принимающий в первом параметре `System.Object`, а во втором — `PaintEventArgs`. Если имеется обработанное событие `Paint` (через дизайнер Visual Studio, или вручную в коде), можно извлечь объект `Graphics` из входящего параметра `PaintEventArgs`. Вот необходимое изменение:

```
public partial class MainForm : Form
{
  public MainForm()
  {
    InitializeComponent();
    CenterToScreen();
    this.Text = "Basic Paint Form";
    // Visual Studio помещает этот код внутрь InitializeComponent().
    this.Paint += new PaintEventHandler(MainForm_Paint);
  }
  private void MainForm_Paint(object sender, PaintEventArgs e)
  {
    Graphics g = e.Graphics;
    g.DrawString("Hello GDI+", new Font("Times New Roman", 20),
       Brushes.Green, 0, 0);
  }
}
```

Независимо от того, как вы реагируете на событие `Paint`, имейте в виду, что всякий раз, когда окно становится "грязным", инициируется событие `Paint`. И как вам должно быть известно, окно считается "грязным" всякий раз, когда изменяется его размер, оно перекрывается другим окном (полностью или частично) и потом открывается, либо сворачивается и затем восстанавливается. Во всех этих случаях .NET гарантирует, что если форма нуждается в перерисовке, обработчик события `Paint` (или переопределенный метод `OnPaint()`) вызывается автоматически.

Объявление клиентской области формы недействительной

Во время работы приложения Windows Forms может понадобиться явно инициировать событие `Paint` в коде, вместо того, чтобы ожидать, пока окно станет "естественно грязным" в результате действий конечного пользователя. Например, можно построить программу, которая позволяет пользователю выбирать из множества предопределенных образов, используя специальное диалоговое окно. После закрытия диалогового окна нужно нарисовать выбранное изображение в клиентской области формы. Очевидно, если вы станете ждать, пока окно станет "естественно грязным", пользователь не увидит изменений до тех пор, пока не изменит размер окна или не перекроет его другим окном. Чтобы программно вынудить окно перерисовать себя, просто вызовите унаследованный метод `Invalidate()`:

```
public partial class MainForm: Form
{
...
  private void MainForm_Paint(object sender, PaintEventArgs e)
  {
    Graphics g = e.Graphics;
    // Здесь визуализировать корректное изображение.
  }
```

```
  private void GetImageFromDialog()
  {
    // Показать диалоговое окно и получить новое изображение.
    // Перерисовать всю клиентскую область.
    Invalidate();
  }
}
```

Метод `Invalidate()` перегружен несколько раз, чтобы позволить специфицировать определенную прямоугольную область для перерисовки вместо перерисовки всей клиентской области окна (что принято по умолчанию). Если вы хотите только обновить самый верхний левый прямоугольник клиентской области, можете написать следующее:

```
// Перерисовать заданную прямоугольную область Form.
private void UpdateUpperArea()
{
  Rectangle myRect = new Rectangle(0, 0, 75, 150);
  Invalidate(myRect);
}
```

Получение объекта `Graphics` вне обработчика события `Paint`

В некоторых редких случаях может понадобиться доступ к объекту `Graphics` за пределами контекста обработчика события `Paint`. Например, предположим, что вы хотите нарисовать маленький кружок в позиции (*x*, *y*), где был выполнен щелчок кнопкой мыши. Один подход к получению действительного объекта `Graphics` из контекста обработчика события `MouseDown` состоит в вызове статического метода `Graphics.FromHwnd()`. Из опыта работы с Win32 вам должно быть известно, что `HWND` — это структура данных, представляющая некоторое окно Win32. На платформе .NET унаследованное свойство `Handle` извлекает лежащий в основе `HWND`, который может применяться в качестве параметра для `Graphics.FromHwnd()`:

```
private void MainForm_MouseDown(object sender, MouseEventArgs e)
{
  // Получить объект Graphics через Hwnd.
  Graphics g = Graphics.FromHwnd(this.Handle);

  // Теперь нарисовать кружок 10*10 в месте щелчка кнопкой мыши.
  g.FillEllipse(Brushes.Firebrick, e.X, e.Y, 10, 10);

  // Освободить все объекты Graphics, созданные непосредственно.
  g.Dispose();
}
```

Хотя эта логика визуализирует кружок вне обработчика события `OnPaint()`, очень важно понимать, что когда форма делается недействительной (и потому должна быть перерисована), каждый из кружков будет стерт! Это имеет смысл, учитывая, что визуализация случается только в контексте события `MouseDown`. Намного лучший подход состоит в том, чтобы позволить обработчику `MouseDown` создать новый экземпляр `Point`, который затем добавляется к внутренней коллекции (вроде `List<T>`), с последующим вызовом `Invalidate()`. В этот момент обработчик событий `Paint` может просто выполнить итерацию по коллекции и нарисовать каждый `Point`:

```
public partial class MainForm : Form
{
  // Используется для запоминания всех точек.
  private List<Point> myPts = new List<Point>();
```

```
  public MainForm()
  {
    ...
    this.MouseDown += new MouseEventHandler(MainForm_MouseDown);
  }
  private void MainForm_MouseDown(object sender, MouseEventArgs e)
  {
    // Добавить в коллекцию точек.
    myPts.Add(new Point(e.X, e.Y));
    Invalidate();
  }
  private void MainForm_Paint(object sender, PaintEventArgs e)
  {
    Graphics g = e.Graphics;
    g.DrawString("Hello GDI+", new Font("Times New Roman", 20),
        new SolidBrush(Color.Black), 0, 0);
    foreach(Point p in myPts)
      g.FillEllipse(Brushes.Firebrick, p.X, p.Y, 10, 10);
  }
}
```

При таком подходе визуализированные кружки навсегда останутся на поверхности окна, поскольку вся графическая визуализация происходит в событии `Paint`. На рис. 3.1 показан результат тестового запуска этого начального приложения GDI+.

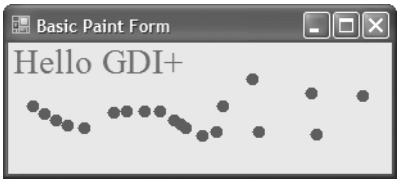

Рис. 3.1. Простое приложение рисования

Исходный код. Проект `BasicPaintForm` включен в подкаталог `Bonus Chapter 3`.

Удаление объекта `Graphics`

Если вы внимательно читали несколько последних страниц, то могли заметить, что в некоторых из приведенных примеров непосредственно вызывался метод `Dispose()` объекта `Graphics`, в то время как в других это не делалось. Учитывая, что тип `Graphics` манипулирует различными неуправляемыми ресурсами, имело бы смысл, чтобы он освобождал ресурсы через вызов `Dispose()` как можно скорее (вместо того, чтобы дожидаться, пока это сделает сборщик мусора в процессе финализации). То же самое можно сказать о любом типе, поддерживающем интерфейс `IDisposable`. При работе с объектами GDI+ `Graphics` следует помнить о следующих правилах:

- если вы напрямую создали объект `Graphics`, вызывайте `Dispose()`, когда надобность в нем отпадет;
- если вы ссылаетесь на существующий объект `Graphics`, *не* вызывайте `Dispose()`.

Чтобы прояснить мысль, рассмотрим следующий обработчик события `Paint`:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
  // Загрузить локальный файл *.jpg.
  Image myImageFile = Image.FromFile("landscape.jpg");

  // Создать объект Graphics на основе изображения.
  Graphics imgGraphics = Graphics.FromImage(myImageFile);

  // Визуализировать новые данные в изображение.
  imgGraphics.FillEllipse(Brushes.DarkOrange, 50, 50, 150, 150);

  // Нарисовать изображение в Form.
  Graphics g = e.Graphics;
  g.DrawImage(myImageFile, new PointF(0.0F, 0.0F));

  // Освободить созданный нами объект Graphics.
  imgGraphics.Dispose();
}
```

Пусть пока вас не волнует, что некоторая логика GDI+ выглядит несколько загадочно. Однако обратите внимание, что вы получили объект `Graphics` из файла `*.jpg`, загруженного из локального каталога (статическим методом `Graphics.FromImage()`). Поскольку вы явно создали этот объект `Graphics`, практический опыт требует вызова метода `Dispose()` на этом объекте по завершении его использования, чтобы освободить внутренние ресурсы для использования другими частями системы.

Обратите внимание, что вы не вызываете `Dispose()` на объекте `Graphics`, который получили из входящего `PaintEventArgs`. Это объясняется тем фактом, что вы не создавали этот объект напрямую, и не можете быть уверены, что он не понадобится другим частям программы. Ясно, что освобождение объекта, используемого потом где-то еще, создало бы проблему!

Кстати, вспомните, что если вы забудете вызвать `Dispose()` на объекте, реализующем `IDisposable`, внутренние ресурсы будут в конечном итоге освобождены, когда позднее до них доберется сборщик мусора. В этом свете ручное освобождение объекта `imgGraphics` технически не обязательно. Хотя явное освобождение объектов GDI+, созданных напрямую — разумная мера, для простоты в примерах кода это делаться с каждым типом GDI+ не будет.

Координатная система GDI+

Наша следующая задача — изучить координатную систему, положенную в основу GDI+. Существуют три системы координат, используемые исполняющей системой для определения местоположения и размера визуализируемого содержимого. Прежде всего, мы имеем то, что называется *мировыми координатами*. Мировые координаты представляют собой абстракцию размера определенного типа GDI+ безотносительно единиц измерения. Например, если вы рисуете прямоугольник с размерами (0, 0, 100, 100), то специфицируете прямоугольник в 100×100 "штук" в размере. Как вы можете догадаться, "штукой" по умолчанию является пиксель; однако можно выбрать и другую единицу измерения (дюйм, сантиметр и т.п.).

Затем мы имеем *страничные координаты*. Страничные координаты представляют смещение, применяемое к исходным мировым координатам. Это удобно, если приходится вручную применять смещения в коде (при необходимости). Например, если имеется форма, которой нужно поддерживать рамку 100×100 пикселей, вы можете специфицировать страничные координаты (100, 100), чтобы позволить всей визуализации начаться с точки (100, 100). Однако в коде можно затем специфицировать простые мировые координаты (избавляясь от необходимости каждый раз вычислять смещение).

И, наконец, существуют *координаты устройства*. Координаты устройства представляют результат применения страничных координат к исходным мировым координатам. Эта координатная система используется для точного определения места, где должен быть визуализирован объект GDI+. Когда вы программируете с применением GDI+, то обычно думаете в терминах мировых координат, которые служат базой для определения размера и местоположения элемента GDI+. Чтобы визуализировать мировые координаты, не требуется никаких усилий по кодированию — просто передайте измерения для текущей операции визуализации:

```
void MainForm_Paint(object sender, PaintEventArgs e)
{
   // Визуализировать прямоугольник в мировых координатах.
   Graphics g = e.Graphics;
   g.DrawRectangle(Pens.Black, 10, 10, 100, 100);
}
```

"За кулисами" мировые координаты автоматически отображаются на страничные координаты, которые затем отображаются на координаты устройства. Во многих случаях напрямую использовать страничные координаты или координаты устройства не придется, если только не захочется применить какие-либо графические трансформации. Учитывая, что предыдущий код не специфицирует никакой логики трансформации, мировые, страничные и координаты устройства здесь совпадают.

Если необходимо применить различные трансформации перед тем, как визуализировать логику GDI+, потребуется использовать различные члены типа `Graphics` (вроде метода `TranslateTransform()`), чтобы специфицировать различные "страничные координаты" относительно существующей мировой координатной системы. В результате получается набор координат устройства, который и будет использоваться для визуализации объекта GDI+ на целевом устройстве:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   // Специфицировать смещение страничных координат (10, 10).
   Graphics g = e.Graphics;
   g.TranslateTransform(10, 10);
   g.DrawRectangle(10, 10, 100, 100);
}
```

В данном случае прямоугольник в действительности визуализируется в левой верхней точке (20, 20), учитывая, что мировые координаты были смещены вызовом `TranslateTransform()`.

Единица измерения по умолчанию

В GDI+ в качестве единицы измерения по умолчанию принят пиксель. Начальная точка располагается в левом верхнем углу, причем ось *x* направлена вправо, а ось *y* — вниз (рис. 3.2).

Таким образом, если вы визуализируете `Rectangle` с использованием красного пера толщиной в 5 пикселей, как показано ниже:

```
void MainForm_Paint(object sender, PaintEventArgs e)
{
   // Установить мировые координаты в единицах измерения по умолчанию.
   Graphics g = e.Graphics;
   g.DrawRectangle(new Pen(Color.Red, 5), 10, 10, 100, 100);
}
```

то увидите квадрат, расположенный на 10 пикселей ниже и правее верхнего левого угла `Form` (рис. 3.3).

Рис. 3.2. Координатная система GDI+ по умолчанию

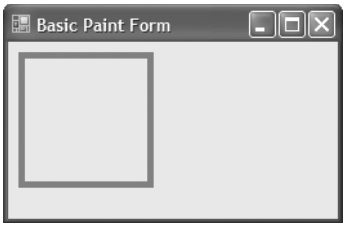

Рис. 3.3. Визуализация в единицах-пикселях

Указание альтернативной единицы измерения

Если не хотите визуализировать изображение с использованием единиц измерения — пикселей, можете изменить эту установку по умолчанию, модифицировав свойство `PageUnit` объекта `Graphics` и тем самым поменяв единицы измерения страничной координатной системы. Свойство `PageUnit` получает любое значение из перечисления `GraphicsUnit`:

```
public enum GraphicsUnit
{
    // Специфицирует мировые координаты.
    World,
    // Пиксели для дисплеев и 1/100 дюйма для принтеров.
    Display,
    // Специфицирует пиксель.
    Pixel,
    // Специфицирует точки принтера (1/72 дюйма).
    Point,
    // Специфицирует дюйм.
    Inch,
    // Специфицирует единицу документа (1/300 дюйма).
    Document,
    // Специфицирует миллиметр.
    Millimeter
}
```

Чтобы проиллюстрировать изменение лежащего в основе `GraphicsUnit`, измените предыдущий код визуализации следующим образом:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
    // Нарисовать прямоугольник в дюймах, а не пикселях.
    Graphics g = e.Graphics;
    g.PageUnit = GraphicsUnit.Inch;
    g.DrawRectangle(new Pen(Color.Red, 5), 0, 0, 100, 100);
}
```

После этого вы увидите радикально другой прямоугольник (рис. 3.4).

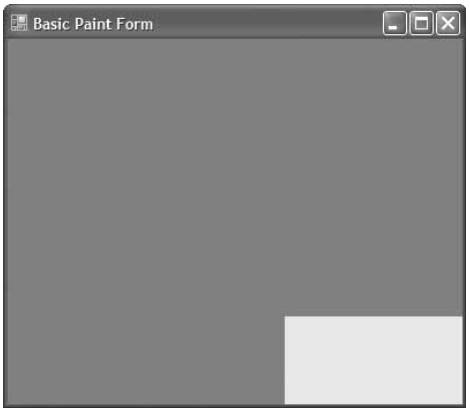

Рис. 3.4. Визуализация в дюймах

Причина того, что почти вся клиентская область Form заполнена красным, состоит в том, что вы сконфигурировали `Pen` толщиной в 5 *дюймов*! Сам прямоугольник теперь имеет размеры 100×100 *дюймов*. Фактически, маленький серый прямоугольник, который вы видите в правом нижнем углу — это левая верхняя часть внутренности прямоугольника.

Указание альтернативной исходной точки

Напомним, что при использовании координатной системы и системы измерения по умолчанию точка (0, 0) находится в самом верхнем левом углу поверхности. Хотя часто это именно то, что нужно, иногда требуется изменить местоположение начала визуализации. Например, предположим, что приложение всегда должно резервировать 100-пиксельную рамку вокруг клиентской области `Form` (по той или иной причине). Необходимо обеспечить, чтобы все операции GDI+ происходили внутри этой области.

Один подход к решению этой задачи состоит в применении вручную смещения ко всему коду визуализации. Это было бы утомительно, поскольку постоянно пришлось бы применять некоторое значение смещения к каждой операции визуализации. Было бы намного лучше (и проще) установить свойство, которое скажет: "Несмотря на указание рисовать прямоугольник в начальной точке (0,0), начинать всегда следует с точки (100,100)". Это бы существенно облегчило работу, поскольку можно было бы специфицировать точки рисования без каких-либо модификаций.

В GDI+ можно сдвинуть начальную точку, установив значение трансформации с помощью метода `TranslateTransform()` класса `Graphics`, позволяющего специфицировать страничную координатную систему, которая будет применена к исходным спецификациям мировых координат. Например:

```
void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   // Установить страничные координаты в (100, 100).
   g.TranslateTransform(100, 100);
   // Мировые координаты по-прежнему (0, 0, 100, 100),
   // однако координаты устройства теперь (100, 100, 200, 200).
   g.DrawRectangle(new Pen(Color.Red, 5), 0, 0, 100, 100);
}
```

Здесь установлены значения мировых координат (0, 0, 100, 100). Однако для страничных координат задано смещение (100, 100). С учетом этого, координаты устройства отображаются на (100, 100, 200, 200). Таким образом, хотя вызов `DrawRectangle()` выглядит так, как если бы вы визуализировали прямоугольник в левом верхнем углу `Form`, на самом деле происходит визуализация, показанная на рис. 3.5.

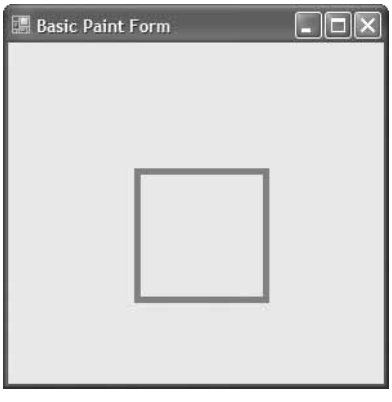

Рис. 3.5. Результат применения смещения страницы

Для экспериментов с некоторыми способами изменения координатной системы GDI+ в загружаемом коде для этой главы имеется пример приложения по имени `CoorSystem`. С помощью двух пунктов меню можно изменять исходную точку, а также единицы измерения (рис. 3.6).

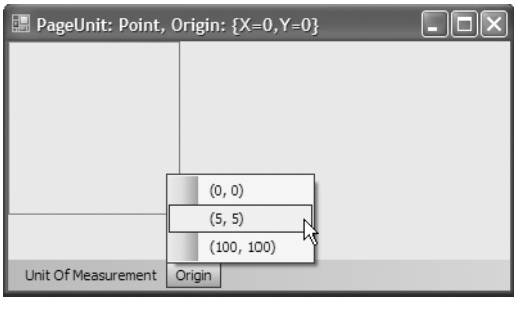

Рис. 3.6. Измерение режимов координатной системы и единиц измерения

Теперь, когда вы лучше понимаете, как применяются внутренние трансформации для определения того, где следует визуализировать данный объект GDI+ на целевом устройстве, давайте подробно рассмотрим манипуляции цветом.

Исходный код. Проект `CoorSystem` включен в подкаталог `Bonus Chapter 3`.

Определение значения цвета

Многие из методов визуализации, определенных в классе `Graphics`, требуют указания цвета, который должен использоваться в процессе рисования. Структура `System.Drawing.Color` представляет цветовую константу ARGB (alpha, red, green, blue — альфа-

канал, красная составляющая, зеленая составляющая, синяя составляющая). Большая часть функциональности типа Color обеспечивается рядом статических свойств, доступных только для чтения, которые возвращают определенный объект типа Color:

```
// Один из многих предопределенных цветов...
Color c = Color.PapayaWhip;
```

Если цветовые значения по умолчанию не подходят, можете создать новый объект Color и специфицировать значения составляющих A, R, G и B в методе FromArgb():

```
// Указание ARGB вручную.
Color myColor = Color.FromArgb(0, 255, 128, 64);
```

Кроме того, с помощью метода FromName() можно сгенерировать объект Color по заданному строковому значению. Символы строкового параметра должны соответствовать одному из членов перечисления KnownColor (включающему значения различных цветовых элементов Windows, такие как KnownColor.WindowFrame и KnownColor.WindowText):

```
// Получить Color по известному имени.
Color myColor = Color.FromName("Red");
```

Независимо от используемого метода, с типом Color можно взаимодействовать через его члены, которые описаны ниже.

- GetBrightness(). Возвращает яркость экземпляра Color на основе показателей "тон-насыщенность-яркость" (hue-saturation-brightness — HSB).
- GetSaturation(). Возвращает насыщенность экземпляра Color на основе показателей HSB.
- GetHue(). Возвращает тон экземпляра Color на основе показателей HSB.
- IsSystemColor. Определяет, представляет ли экземпляр Color системный цвет.
- A, R, G, B. Возвращает значение, присвоенное составляющим ARGB экземпляра Color.

Класс ColorDialog

Для того чтобы позволить конечному пользователю приложения самостоятельно конфигурировать экземпляр типа Color, в пространстве имен System.Windows.Forms предусмотрено предопределенное диалоговое окно по имени ColorDialog (рис. 3.7).

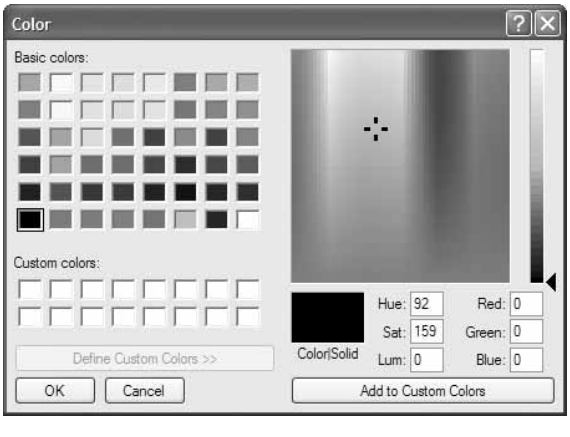

Рис. 3.7. Диалоговое окно выбора цвета Windows Forms

Работать с этим диалоговым окном достаточно просто. Используя действительный экземпляр типа ColorDialog, вызовите ShowDialog() для отображения его в модальном режиме. Как только пользователь закроет это диалоговое окно, вы сможете извлечь соответствующий объект Color из свойства ColorDialog.Color.

Предположим, что необходимо позволить пользователю конфигурировать цвет фона клиентской области Form с помощью ColorDialog. Для простоты ColorDialog будет отображаться, когда пользователь щелкнет в любом месте клиентской области:

```
public partial class MainForm : Form
{
  private ColorDialog colorDlg;
  private Color currColor = Color.DimGray;
  public MainForm()
  {
    InitializeComponent();
    colorDlg = new ColorDialog();
    Text = "Click on me to change the color";
    this.MouseDown += new MouseEventHandler(MainForm_MouseDown);
  }
  private void MainForm_MouseDown(object sender, MouseEventArgs e)
  {
    if (colorDlg.ShowDialog() != DialogResult.Cancel)
    {
      currColor = colorDlg.Color;
      this.BackColor = currColor;
      string strARGB = colorDlg.Color.ToString();
      MessageBox.Show(strARGB, "Color is:");
    }
  }
}
```

Исходный код. Приложение ColorDlg включено в подкаталог Bonus Chapter 3.

Управление шрифтами

Теперь давайте разберемся, как программно манипулировать шрифтами. Тип System.Drawing.Font представляет определенный шрифт, инсталлированный на машине пользователя. Типы шрифтов могут определяться с использованием любого из перегруженных конструкторов. Вот несколько примеров:

```
// Создать Font по заданному имени типа и размеру.
Font f = new Font("Times New Roman", 12);

// Создать Font по заданному имени, типу и стилю.
Font f2 = new Font("WingDings", 50, FontStyle.Bold | FontStyle.Underline);
```

Здесь f2 создается применением логической операции “ИЛИ” к набору значение перечисления FontStyle:

```
public enum FontStyle
{
   Regular, Bold,
   Italic, Underline, Strikeout
}
```

После настройки внешнего вида и поведения объекта Font следующей задачей будет его передача в виде параметра методу Graphucs.DrawString(). Хотя DrawString() также перегружен несколько раз, каждая вариация обычно требует одной и той же ба-

зовой информации: текст для отображения, шрифт, в котором его надо показать, кисть, используемая для визуализации, и местоположение, куда его поместить.

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;

   // Специфицировать (String, Font, Brush, Point) как аргументы.
   g.DrawString("My string", new Font("WingDings", 25),
   Brushes.Black, new Point(0,0));

   // Специфицировать (String, Font, Brush, int, int)
   g.DrawString("Another string", new Font("Times New Roman", 16),
      Brushes.Red, 40, 40);
}
```

Работа с семействами шрифтов

В пространстве имен `System.Drawing` также определен тип `FontFamily`, который абстрагирует группу гарнитур, имеющих сходный базовый дизайн, но различные вариации стиля. Семейство шрифтов, такое как Verdana, может включать несколько шрифтов, отличающихся по стилю и размеру. Например, Verdana 12-point bold и Verdana 24-point italic — разные шрифты одного семейства Verdana.

Конструктор типа `FontFamily` принимает строковое представление имени семейства шрифтов, которое вы пытаетесь получить. После создания "общего семейства" можно создавать более специфичные объекты `Font`:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   // Создать семейство шрифтов.
   FontFamily myFamily = new FontFamily("Verdana");
   // Передать семейство конструктору Font.
   Font myFont = new Font(myFamily, 12);
   g.DrawString("Hello!", myFont, Brushes.Blue, 10, 10);
}
```

Наибольший интерес представляет возможность сбора статистики относительно заданного семейства шрифтов. Например, пусть вы строите приложение обработки текстов и хотите определить среднюю ширину символа в определенном `FontFamily`. Что, если необходимо знать значения верхнего и нижнего выносных элементов определенного символа? Для ответов на подобные вопросы в типе `FontFamily` определены ключевые члены, которые перечислены в табл. 3.5.

Таблица 3.5. Члены типа `FontFamily`

Член	Назначение
`GetCellAscent()`	Возвращает метрику верхнего выносного элемента для членов семейства шрифтов.
`GetCellDescent()`	Возвращает метрику нижнего выносного элемента для членов семейства шрифтов.
`GetLineSpacing()`	Возвращает расстояние между двумя строками символов текста для данного семейства `FontFamily` с указанным `FontStyle`.
`GetName()`	Возвращает имя `FontFamily` на указанном языке.
`IsStyleAvailable()`	Указывает доступность указанного семейства `FontFamily`.

Для целей иллюстрации ниже приведен обработчик события `Paint`, который печатает ряд характеристик семейства шрифтов Verdana:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   FontFamily myFamily = new FontFamily("Verdana");
   Font myFont = new Font(myFamily, 12);
   int y = 0;
   int fontHeight = myFont.Height;

   // Показать единицы измерения членов FontFamily.
   this.Text = "Measurements are in GraphicsUnit." + myFont.Unit;
   g.DrawString("The Verdana family.", myFont, Brushes.Blue, 10, y);
   y += 20;

   // Распечатать 'семейные узы'...
   g.DrawString("Ascent for bold Verdana: " +
             myFamily.GetCellAscent(FontStyle.Bold),
             myFont, Brushes.Black, 10, y + fontHeight);
   y += 20;
   g.DrawString("Descent for bold Verdana: " +
             myFamily.GetCellDescent(FontStyle.Bold),
             myFont, Brushes.Black, 10, y + fontHeight);
   y += 20;
   g.DrawString("Line spacing for bold Verdana: " +
             myFamily.GetLineSpacing(FontStyle.Bold),
             myFont, Brushes.Black, 10, y + fontHeight);
   y += 20;
   g.DrawString("Height for bold Verdana: " +
             myFamily.GetEmHeight(FontStyle.Bold),
             myFont, Brushes.Black, 10, y + fontHeight);
   y += 20;
}
```

Результат показан на рис. 3.8.

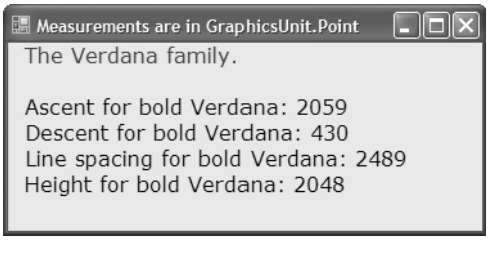

Рис. 3.8. Сбор статистики о семействе шрифтов Verdana

Обратите внимание, что эти члены типа `FontFamily` возвращают значения, используя `GraphicsUnit.Point` (а не `Pixel`) в качестве единицы измерения, что соответствует 1/72 дюйма. Можете трансформировать эти значения в другие единицы измерения по своему усмотрению.

Исходный код. Приложение `FontFamilyApp` включено в подкаталог `Bonus Chapter 3`.

Работа с начертаниями и размерами шрифта

Далее мы построим более сложное приложение, которое позволит пользователю манипулировать объектом `Font`, поддерживаемым `Form`. Приложение даст возможность выбирать текущее начертание шрифта из предопределенного набора, используя пункт меню Configure⇨Font Face (Конфигурирование⇨Начертание шрифта). Также можно будет неявно управлять размером объекта `Font`, используя объект Windows Forms под названием `Timer`. Если пользователь активизирует `Timer` через пункт меню Configure⇨Swell? (Конфигурирование ⇨Нарастать?), то размер объекта `Font` начнет расти с регулярным интервалом (вплоть до верхнего предела). Таким образом, текст начнет "разбухать", создавая анимацию "дышащего" текста. И, наконец, последний подэлемент меню Configure под названием List Installed Fonts (Вывести список инсталлированных шрифтов), будет служить для вывода списка всех шрифтов, инсталлированных на машине конечного пользователя. На рис. 3.9 показана компоновка меню.

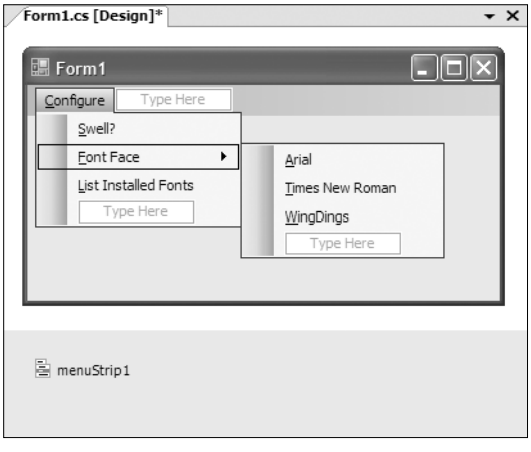

Рис. 3.9. Компоновка меню проекта `FontApp`

Для начала реализации приложения модифицируйте `Form`, добавив переменную-член типа `Timer` (по имени `swellTimer`), строки (`strFontFace`) для представления текущего начертания шрифта и целочисленного значения (`swellValue`) для представления степени изменения размера шрифта. Внутри конструктора `Form` сконфигурируйте `Timer` для выдачи события `Tick` каждые 100 миллисекунд:

```
public partial class MainForm : Form
{
  private Timer swellTimer = new Timer();
  private int swellValue;
  private string strFontFace = "WingDings";
  public MainForm()
  {
    InitializeComponent();
    BackColor = Color.Honeydew;
    CenterToScreen();
    // Сконфигурировать Timer.
    swellTimer.Enabled = true;
    swellTimer.Interval = 100;
    swellTimer.Tick += new EventHandler(swellTimer_Tick);
  }
}
```

В обработчике события Tick значение члена данных swellValue увеличивается на 5. Напомним, что swellValue будет прибавлено к текущему размеру шрифта для реализации простой анимации (предположим, что swellValue может иметь предельное значение 50). Чтобы сократить мерцание, которое может происходить при перерисовке всей клиентской области, метод Invalidate() вызывается только для обновления верхней прямоугольной области Form:

```
private void swellTimer_Tick(object sender, EventArgs e)
{
  // Увеличить текущее значение swellValue на 5.
  swellValue += 5;
  // Если значение больше или равно 50, сбросить его до нуля.
  if(swellValue >= 50)
    swellValue = 0;
  // Объявить недействительным минимальный прямоугольник для сокращения мерцания.
  Invalidate(new Rectangle(0, 0, ClientRectangle.Width, 100));
}
```

Теперь, когда 100 верхних пикселей клиентской области обновляются при каждом событии Timer, надо что-нибудь визуализировать. В обработчике Paint формы создайте объект Font на основе определенного пользователем начертания (выбранного из пункта меню) и текущего значения swellValue (продиктованного Timer). Имея полностью сконфигурированный объект Font, визуализируйте сообщение в середине прямоугольника, который объявляется недействительным:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   // Размер нашего шрифта может быть между 12 и 62,
   // в зависимости от текущего значения swellValue.
   Font theFont = new Font(strFontFace, 12 + swellValue);
   string message = "Hello GDI+";
   // Отобразить сообщение в центре прямоугольника.
   float windowCenter = this.DisplayRectangle.Width/2;
   SizeF stringSize = g.MeasureString(message, theFont);
   float startPos = windowCenter - (stringSize.Width/2);
   g.DrawString(message, theFont, new SolidBrush(Color.Blue), startPos, 10);
}
```

Как можно было догадаться, если пользователь выбирает определенное начертание шрифта, обработчик события Clicked каждого пункта меню должен обновлять строковую переменную fontFace и объявлять недействительной клиентскую область. Например:

```
private void arialToolStripMenuItem_Click(object sender, EventArgs e)
{
   strFontFace = "Arial";
   Invalidate();
}
```

Обработчик события Click пункта меню Swell будет использоваться для того, чтобы позволить пользователю останавливать и запускать увеличение текста (т.е. включать и отключать анимацию). Для этого переключайте свойство Enabled переменной Timer следующим образом:

```
private void swellToolStripMenuItem_Click(object sender, EventArgs e)
{
   swellTimer.Enabled = !swellTimer.Enabled;
}
```

Перечисление инсталлированных шрифтов

Теперь давайте расширим приложение, добавив возможность отображения установленного на целевой машине множества шрифтов с использованием типов из `System.Drawing.Text`. Это пространство имен содержит группу типов, которые можно применять для исследования и манипуляций набором шрифтов, инсталлированных на целевой машине. Для наших целей рассмотрим только класс `InstalledFontCollection`.

Когда пользователь выбирает пункт меню Configure⇨List Installed Fonts, соответствующий обработчик события `Clicked` создаст экземпляр класса `InstalledFontCollection`. Этот класс поддерживает массив по имени `FontFamily`, который представляет набор всех шрифтов на целевой машине, и может быть получен из свойства `InstalledFontCollection.Families`. С помощью свойства `FontFamily.Name` можно извлекать начертание шрифта (например, Times New Roman, Arial и т.п.) для каждого шрифта.

Добавьте в класс `Form` строковый член данных по имени `installedFonts` для хранения всех начертаний шрифтов. Логика обработчика меню List Installed Fonts создает экземпляр типа `InstalledFontCollection`, читает имя каждого шрифта и добавляет новое начертание шрифта в приватную переменную-член `installedFonts`.

```
public partial class MainForm : Form
{
  // Для хранения списка шрифтов.
  private string installedFonts;
  // Обработчик для получения списка инсталлированных шрифтов.
  private void mnuConfigShowFonts_Clicked(object sender, EventArgs e)
  {
    InstalledFontCollection fonts = new InstalledFontCollection();
    for(int i = 0; i < fonts.Families.Length; i++)
      installedFonts += fonts.Families[i].Name + " ";
    // На этот раз нам нужно объявить недостоверной всю клиентскую область,
    // поскольку мы выведем строку installedFonts в нижней половине
    // клиентского прямоугольника.
    Invalidate();
  }
...
}
```

Последняя задача — визуализация строки `installedFonts` в клиентской области, непосредственно под областью экрана, где отображается наш "дышащий" текст:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   Font theFont = new Font(strFontFace, 12 + swellValue);
   string message = "Hello GDI+";
   // Отобразить сообщение в центре окна.
   float windowCenter = this.DisplayRectangle.Width/2;
   SizeF stringSize = e.Graphics.MeasureString(message, theFont);
   float startPos = windowCenter - (stringSize.Width/2);
   g.DrawString(message, theFont, Brushes.Blue, startPos, 10);
   // Ниже показать список инсталлированных шрифтов.
   Rectangle myRect = new Rectangle(0, 100,
      ClientRectangle.Width, ClientRectangle.Height);
   // Вывести надпись черным цветом в этой области Form.
   g.FillRectangle(new SolidBrush(Color.Black), myRect);
   g.DrawString(installedFonts, new Font("Arial", 12),
      Brushes.White, myRect);
}
```

Напомним, что размер прямоугольника, объявляемого недопустимым, отображен на верхние 100 пикселей клиентского прямоугольника. Поскольку обработчик `Tick` объявляет недопустимой только часть `Form`, оставшаяся область не перерисовывается в ответ на событие `Tick` (чтобы оптимизировать визуализацию клиентской области).

В качестве последнего штриха для обеспечения правильной перерисовки давайте обработаем событие `Resize` класса формы, чтобы нижняя часть клиентского прямоугольника была перерисована правильно:

```
private void MainForm_Resize(object sender, System.EventArgs e)
{
    Rectangle myRect = new Rectangle(0, 100,
        ClientRectangle.Width, ClientRectangle.Height);
    Invalidate(myRect);
}
```

На рис. 3.10 показан результат (с текстом, выведенным в шрифте Wingdings).

Исходный код. Проект `SwellingFontApp` включен в подкаталог `Bonus Chapter 3`.

Класс `FontDialog`

Как и можно было предположить, существует готовое диалоговое окно выбора шрифта (`FontDialog`), которое показано на рис. 3.11.

Подобно типу `ColorDialog`, рассмотренному ранее в этой главе, для работы с `FontDialog` необходимо просто вызвать метод `ShowDialog()`. С помощью свойства `Font` можно извлечь характеристики текущего выбора для использования в приложении. Когда пользователь щелкает в любом месте формы, отображается диалоговое окно выбора шрифта и затем визуализируется сообщение о текущем выбранном шрифте:

Рис. 3.10. Приложение `FontApp` в действии

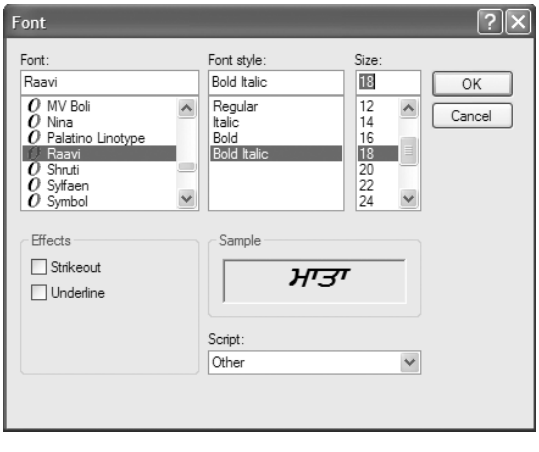

Рис. 3.11. Диалоговое окно выбора шрифта Windows Forms

```
public partial class MainForm : Form
{
  private FontDialog fontDlg = new FontDialog();
  private Font currFont = new Font("Times New Roman", 12);
  public MainForm()
  {
    InitializeComponent();
    CenterToScreen();
  }
  private void MainForm_MouseDown(object sender, MouseEventArgs e)
  {
    if (fontDlg.ShowDialog() != DialogResult.Cancel)
    {
      currFont = fontDlg.Font;
      this.Text = string.Format("Selected Font: {0}", currFont);
      Invalidate();
    }
  }
  private void MainForm_Paint(object sender, PaintEventArgs e)
  {
    Graphics g = e.Graphics;
    g.DrawString("Testing...", currFont, Brushes.Black, 0, 0);
  }
}
```

Исходный код. Приложение `FontDlgForm` включено в подкаталог `Bonus Chapter 3`.

Обзор пространства имен `System.Drawing.Drawing2D`

Теперь, освоив манипулирование типами `Font`, следующей задачей будет научиться работать с объектами `Pen` и `Brush` для визуализации графических изображений. Хотя это можно делать посредством только вспомогательных типов `Brushes` и `Pens`, чтобы получать предварительно сконфигурированные типы в сплошном цвете, следует знать о существовании более "экзотических" перьев и кистей в пространстве имен `System.Drawing.Drawing2D`.

Это дополнительное пространство имен GDI+ предоставляет ряд классов, позволяющих модифицировать концы линий (треугольник, ромб, и т.п.) конкретного пера, строить текстурированные кисти и работать с векторной графикой. Некоторые из основных типов этого пространства имен, о которых следует знать, перечислены в табл. 3.6 (они сгруппированы по функциональности).

Работа с перьями

Типы GDI+ `Pen` используются для рисования линий между двумя конечными точками. Однако `Pen` сам по себе мало чего стоит. Когда необходимо визуализировать геометрическую фигуру на элементе-наследнике типа `Control`, действительный тип `Pen` передается любому количеству методов визуализации, определенных классом `Graphics`. В общем случае, методы `Draw`*`XXX`*`()` служат для визуализации некоторого набора линий на графической поверхности и обычно используются с объектами `Pen`.

Тип `Pen` определяет небольшой набор конструкторов, которые позволяют определять начальный цвет и ширину штриха данного пера. Большая часть функциональности `Pen` происходит от поддерживаемых им свойств. Частичный список представлен в табл. 3.7.

Таблица 3.6. Классы из пространства `System.Drawing.Drawing2D`

Классы	Назначение
`AdjustableArrowCap` `CustomLineCap`	Концы перьев для рисования начальных и конечных точек линии. Эти типы представляют изменяемые стрелочные и определяемые пользователем концы.
`Blend` `ColorBlend`	Эти классы используются для определения шаблона смешения (и цветов), используемого в сочетании с `LinearGradientBrush`.

`GraphicsPath`
`GraphicsPathIterator`
`PathData`
	Объект `GraphicsPath` представляет серию отрезков прямых и кривых. Этот класс позволяет вставлять в путь почти любую из графических элементов (дуги, прямоугольники, линии, строки, многоугольники и т.п.). `PathData` хранит графические данные, составляющие путь.

`HatchBrush`
`LinearGradientBrush`
`PathGradientBrush`
	Экзотические типы кистей.

Таблица 3.7. Свойства `Pen`

Свойства	Назначение
`Brush`	Определяет кисть, используемую пером.
`Color`	Определяет цвет, используемый пером.
`CustomStartCap` `CustomEndCap`	Получает или устанавливает стиль концов линий, нарисованных данным пером. *Стиль конца* (cap style) — термин, используемый для описания вида начала и конца штриха данного экземпляра `Pen`. Эти свойства позволяют строить специальные концы для ваших экземпляров `Pen`.
`DashCap`	Получает или устанавливает стиль конца, используемый в штриховых линиях, нарисованных данным `Pen`.
`DashPatterm`	Получает или устанавливает массив штрихов и пробелов. Штрихи состоят из сегментов линий.
`DashStyle`	Получает или устанавливает стиль, используемый для пунктирных линий, нарисованных данным `Pen`.
`StartCap` `EndCap`	Получает или устанавливает предопределенный стиль концов линий, нарисованных данным `Pen`. Конец `Pen` определяется с использованием перечисления `LineCap`, определенного в пространстве имен `System.Drawing.Drawing2D`.
`Width`	Получает или устанавливает ширину `Pen`.
`DashOffset`	Получает или устанавливает расстояние от начала линии до начала шаблона штрихов.

Помните, что в дополнение к типу `Pen` GDI+ предлагает коллекцию `Pens`. Используя множество статических свойств, можно извлекать `Pen` (заданного цвета) на лету, вместо того, чтобы создавать специальный `Pen` вручную. Однако имейте в виду, что возвращаемые экземпляры `Pen` всегда имеют ширину 1.

Если требуется более экзотическое перо, придется строить его вручную. Давайте визуализируем некоторые графические изображения с использованием простых типов `Pen`. Предположим, что есть главный объект `Form`, который может реагировать на запросы рисования. Реализация будет такой:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;

    // Создать синее перо.
    Pen bluePen = new Pen(Color.Blue, 20);

    // Получить готовое перо из Pens.
    Pen pen2 = Pens.Firebrick;

    // Визуализировать некоторую графику этими перьями.
    g.DrawEllipse(bluePen, 10, 10, 100, 100);
    g.DrawLine(pen2, 10, 130, 110, 130);
    g.DrawPie(Pens.Black, 150, 10, 120, 150, 90, 80);

    // Нарисовать пурпурный пунктирный многоугольник...
    Pen pen3 = new Pen(Color.Purple, 5);
    pen3.DashStyle = DashStyle.DashDotDot;
    g.DrawPolygon(pen3, new Point[]{new Point(30, 140),
        new Point(265, 200), new Point(100, 225),
        new Point(190, 190), new Point(50, 330),
        new Point(20, 180)});

    // Добавить прямоугольник с некоторым текстом...
    Rectangle r = new Rectangle(150, 10, 130, 60);
    g.DrawRectangle(Pens.Blue, r);
    g.DrawString("Hello out there...How are ya?",
        new Font("Arial", 12), Brushes.Black, r);
}
```

Обратите внимание, что `Pen`, применяемый для визуализации многоугольника, использует перечисление `DashStyle` (определенное в пространстве имен `System.Drawing.Drawing2D`):

```
public enum DashStyle
{
    Solid, Dash, Dot,
    DashDot, DashDotDot, Custom
}
```

В дополнение к предварительно сконфигурированным стилям `DashStyle`, можно определять собственные шаблоны с помощью свойства `DashPattern` типа `Pen`:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
...
    // Рисовать специальный пунктирный шаблон по всей границе формы.
    Pen customDashPen = new Pen(Color.BlueViolet, 10);
    float[] myDashes = { 5.0f, 2.0f, 1.0f, 3.0f };
    customDashPen.DashPattern = myDashes;
    g.DrawRectangle(customDashPen, ClientRectangle);
}
```

На рис. 3.12 показан конечный вывод этого обработчика события `Paint`.

Исходный код. Проект `CustomPenApp` включен в подкаталог `Bonus Chapter 3`.

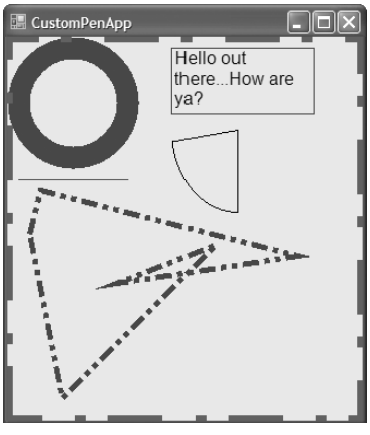

Рис. 3.12. Работа с типами `Pen`

Работа с концами перьев

Если вы посмотрите на вывод предыдущего примера, то заметите, что начала и концы каждой линии визуализировались с использованием стандартного пера (торец имеет прямоугольную форму). Однако с помощью перечисления `LineCap` можно строить перья с более изощренными концами:

```
public enum LineCap
{
   Flat, Square, Round,
   Triangle, NoAnchor,
   SquareAnchor, RoundAnchor,
   DiamondAnchor, ArrowAnchor,
   AnchorMask, Custom
}
```

Для иллюстрации в следующем приложении `PenCapApp` рисуются серии линий с применением всех перечисленных в `LineCap` стилей. Конечный результат показан на рис. 3.13.

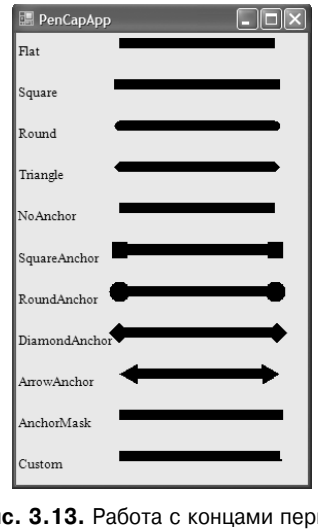

Рис. 3.13. Работа с концами перьев

Код просто проходит циклом по членам перечисления LineCap и печатает имя элемента (например, ArrowAnchor). Затем он настраивает и рисует линию с текущим концом:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   Pen thePen = new Pen(Color.Black, 10);
   int yOffSet = 10;
   // Получить все члены перечисления LineCap.
   Array obj = Enum.GetValues(typeof(LineCap));
   // Нарисовать линию с текущим членом LineCap.
   for(int x = 0; x < obj.Length; x++)
   {
      // Получить следующий конец и конфигурировать перо.
      LineCap temp = (LineCap)obj.GetValue(x);
      thePen.StartCap = temp;
      thePen.EndCap = temp;
      // Напечатать имя текущего члена перечисления LineCap.
      g.DrawString(temp.ToString(), new Font("Times New Roman", 10),
                   new SolidBrush(Color.Black), 0, yOffSet);
      // Нарисовать линию с текущим концом.
      g.DrawLine(thePen, 100, yOffSet, Width - 50, yOffSet);
      yOffSet += 40;
   }
}
```

Исходный код. Проект PenCapApp включен в подкаталог Bonus Chapter 3.

Работа с кистями

Типы-наследники System.Drawing.Brush используются для заполнения области заданным цветом, шаблоном или изображением. Сам класс Brush — абстрактный тип, экземпляры которого не могут быть созданы непосредственно. Однако Brush служит базовым классом для других связанных с ним типов (например, SolidBrush, HatchBrush, LinearGradientBrush и т.д.). В дополнение к специфическим типам-наследникам Brush, пространство имен System.Drawing также определяет два вспомогательных класса, которые возвращают сконфигурированную кисть, используя для этого ряд статических свойств: Brushes и SystemBrushes. В любом случае, как только кисть определена, можно вызывать любые методы Fill*XXX*() типа Graphics.

Довольно интересно то, что на основе заданной кисти также можно строить специальные экземпляры Pen. Таким образом, можно создать необходимую кисть (например, кисть, которая рисует битовое изображение) и визуализировать некоторые геометрические шаблоны с настроенным экземпляром Pen. Для иллюстрации рассмотрим небольшой пример программы, в которой применяются различные экземпляры Brush:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   // Создать SolidBrush синего цвета.
   SolidBrush blueBrush = new SolidBrush(Color.Blue);
   // Получить готовую кисть от типа Brushes.
   SolidBrush pen2 = (SolidBrush)Brushes.Firebrick;
   // Визуализировать некоторые фигуры полученными кистями.
   g.FillEllipse(blueBrush, 10, 10, 100, 100);
   g.FillPie(Brushes.Black, 150, 10, 120, 150, 90, 80);
```

```
    // Нарисовать пурпурный многоугольник...
    SolidBrush brush3= new SolidBrush(Color.Purple);
    g.FillPolygon(brush3, new Point[]{ new Point(30, 140),
       new Point(265, 200), new Point(100, 225),
       new Point(190, 190), new Point(50, 330), new Point(20, 180)} );
    // И прямоугольник с некоторым текстом...
    Rectangle r = new Rectangle(150, 10, 130, 60);
    g.FillRectangle(Brushes.Blue, r);
    g.DrawString("Hello out there...How are ya?",
    new Font("Arial", 12), Brushes.White, r);
}
```

Если вы заметили, это приложение несколько больше, чем `CustomPenApp`, так как на этот раз используются методы `FillXXX()` и типы `SolidBrush` вместо перьев и связанных методов `DrawXXX()`. На рис. 3.14 показан вывод.

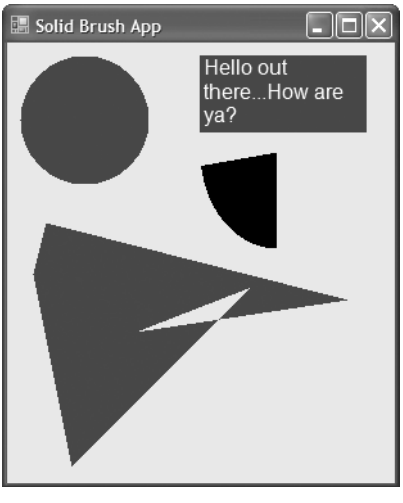

Рис. 3.14. Работа с типами `Brush`

Исходный код. Проект `SolidBrushApp` включен в подкаталог `Bonus Chapter 3`.

Работа с `HatchBrush`

Пространство имен `System.Drawing.Drawing2D` определяет тип-наследник `Brush` под названием `HatchBrush`. Этот тип позволяет заполнить область, используя (огромный) набор предварительно определенных шаблонов, представленных перечислением `HatchStyle`. Вот неполный список их наименований:

```
public enum HatchStyle
{
   Horizontal, Vertical, ForwardDiagonal,
   BackwardDiagonal, Cross, DiagonalCross,
   LightUpwardDiagonal, DarkDownwardDiagonal,
   DarkUpwardDiagonal, LightVertical,
   NarrowHorizontal, DashedDownwardDiagonal,
   SmallConfetti, LargeConfetti, ZigZag,
   Wave, DiagonalBrick, Divot, DottedGrid, Sphere,
   OutlinedDiamond, SolidDiamond,
...
}
```

При конструировании HatchBrush необходимо специфицировать цвета фона и переднего плана для использования в операции заполнения. Для иллюстрации давайте переделаем логику, использованную ранее в примере PenCapApp:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    int yOffSet = 10;

    // Получить все члены перечисления HatchStyle.
    Array obj = Enum.GetValues(typeof(HatchStyle));

    // Нарисовать овал, используя первые 5 типов HatchStyle.
    for (int x = 0; x < 5; x++)
    {
        // Сконфигурировать кисть.
        HatchStyle temp = (HatchStyle)obj.GetValue(x);
        HatchBrush theBrush = new HatchBrush(temp,
            Color.White, Color.Black);

        // Напечатать имя текущего члена перечисления HatchStyle.
        g.DrawString(temp.ToString(), new Font("Times New Roman", 10),
            Brushes.Black, 0, yOffSet);

        // Закрасить прямоугольник текущей кистью.
        g.FillEllipse(theBrush, 150, yOffSet, 200, 25);
        yOffSet += 40;
    }
}
```

Вывод визуализирует закрашенный овал для первых пяти значений перечисления HatchStyle (рис. 3.15).

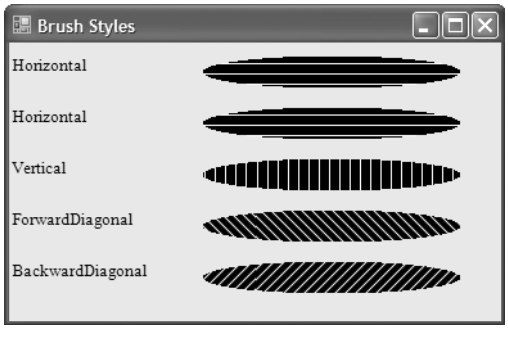

Рис. 3.15. Избранные стили штриховки

Исходный код. Приложение BrushStyles включено в подкаталог Bonus Chapter 3.

Работа с TextureBrush

Тип TextureBrush позволяет прикреплять к кисти битовые изображения, которые затем могут быть использованы в сочетании с операциями заполнения. Очень скоро вы подробно ознакомитесь с устройством класса GDI+ Image. А пока знайте, что TextureBrush присваивается ссылка на Image для использования на протяжении его времени жизни. Само изображение обычно хранится в некотором локальном файле (*.bmp, *.gif, *.jpg) либо встроено в сборку .NET.

Давайте построим пример приложения, в котором применяется тип TextureBrush. Одна кисть будет использоваться для заполнения всей клиентской области изображением из файла по имени clouds.bmp, а другая — для рисования текста с изображением из файла soap bubbles.bmp. Вывод показан на рис. 3.16.

Рис. 3.16. Битовые изображения в качестве кистей

Для начала класс-наследник Form будет поддерживать переменные-члены типа Brush, которым в конструкторе присваиваются ссылки на новые экземпляры TextureBrush. Обратите внимание, что конструктор TextureBrush требует типа, производного от Image:

```
public partial class MainForm : Form
{
  // Данные для кистей-изображений.
  private Brush texturedTextBrush;
  private Brush texturedBGroundBrush;
  public MainForm()
  {
    ...
    // Загрузить изображение для фоновой кисти.
    Image bGroundBrushImage = new Bitmap("Clouds.bmp");
    texturedBGroundBrush = new TextureBrush(bGroundBrushImage);
    // Загрузить изображение для текстовой кисти.
    Image textBrushImage = new Bitmap("Soap Bubbles.bmp");
    texturedTextBrush = new TextureBrush(textBrushImage);
  }
}
```

На заметку! Файлы *.bmp, используемые в этом примере, должны находиться в той же папке, что и приложение (или указаны жестко закодированными путями). Позднее в этой главе мы преодолеем это ограничение.

Имея два экземпляра TextureBrush для визуализации, написать обработчик события Paint достаточно просто:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   Rectangle r = ClientRectangle;
   // Нарисовать облака в клиентской области.
   g.FillRectangle(texturedBGroundBrush, r);
   // Некоторый крупный полужирный текст, нарисованный текстурированной кистью.
   g.DrawString("Bitmaps as brushes! Way cool...", new Font("Arial", 30,
       FontStyle.Bold | FontStyle.Italic), texturedTextBrush, r);
}
```

Исходный код. Приложение TexturedBrushes включено в подкаталог Bonus Chapter 3.

Работа с LinearGradientBrush

И последний тип кистей, который мы рассмотрим — LinearGradientBrush, который можно применять всякий раз, когда необходим градиентный переход между двумя цветами. Работать с этим типом не сложнее, чем с другими типами кистей. Единственное, что здесь представляет интерес, так это то, что при построении LinearGradientBrush понадобится специфицировать пару экземпляров Color и направление перехода через перечисление LinearGradientMode:

```
public enum LinearGradientMode
{
   Horizontal, Vertical,
   ForwardDiagonal, BackwardDiagonal
}
```

Чтобы проверить каждое значение, давайте визуализируем серии прямоугольников, используя LinearGradientBrush:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   Rectangle r = new Rectangle(10, 10, 100, 100);
   // Градиентная кисть.
   LinearGradientBrush theBrush = null;
   int yOffSet = 10;
   // Получить все члены перечисления LinearGradientMode.
   Array obj = Enum.GetValues(typeof(LinearGradientMode));
   // Нарисовать овал с использованием члена LinearGradientMode.
   for(int x = 0; x < obj.Length; x++)
   {
     // Сконфигурировать кисть.
     LinearGradientMode temp = (LinearGradientMode)obj.GetValue(x);
     theBrush = new LinearGradientBrush(r, Color.GreenYellow, Color.Blue, temp);
     // Напечатать имя члена перечисления LinearGradientMode.
     g.DrawString(temp.ToString(), new Font("Times New Roman", 10),
        new SolidBrush(Color.Black), 0, yOffSet);
     // Закрасить прямоугольник текущей кистью.
     g. FillRectangle(theBrush, 150, yOffSet, 200, 50);
     yOffSet += 80;
   }
}
```

На рис. 3.17 показан конечный результат.

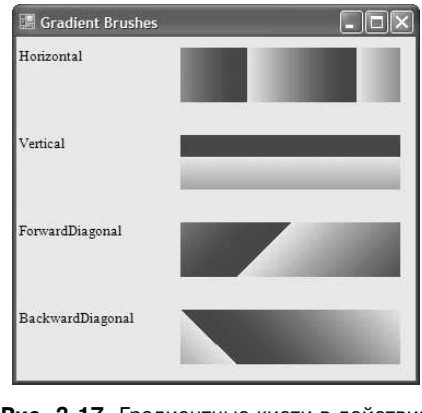

Рис. 3.17. Градиентные кисти в действии

Исходный код. Приложение `GradientBrushes` включено в подкаталог `Bonus Chapter 3`.

Визуализация изображений

К настоящему моменту вы научились манипулировать тремя из четырех основных типов GDI+: шрифтами, перьями и кистями. Последний тип, который мы рассмотрим в этой главе — класс `Image` и связанные с ним подтипы. Абстрактный тип `System.Drawing.Image` определяет множество методов и свойств, которые хранят различную информацию о представленных им графических данных. Например, класс `Image` предоставляет свойства `Width`, `Height` и `Size` для извлечения размеров изображения. Другие свойства позволяют получить доступ к лежащей в основе изображения палитре. Класс `Image` определяет основные члены, перечисленные в табл. 3.8.

Таблица 3.8. Члены типа `Image`

Члены	Назначение
`FromFile()`	Этот статический метод создает `Image` из указанного файла.
`FromStream()`	Этот статический метод создает `Image` из указанного потока данных.

`Height`
`Width`
`Size`
`HorizontalResolution`
`VerticalResolution`
	Эти свойства возвращают информацию о размерах данного `Image`.

`Palette`	Это свойство возвращает тип данных `ColorPalette`, который представляет палитру, лежащую в основе данного `Image`.
`GetBounds()`	Этот метод возвращает `Rectangle`, представляющий текущий размер данного `Image`.
`Save()`	Этот метод сохраняет в файл данные, хранящиеся в объекте-наследнике `Image`.

Учитывая, что создавать экземпляры абстрактного класса `Image` напрямую невозможно, обычно будут создаваться экземпляры типа `Bitmap`. Предположим, что некоторый класс-наследник `Form` визуализирует три битовых изображения в своей клиентской области. Наполнив типы `Bitmap` содержимым корректного файла изображения, просто визуализируйте каждый из них внутри обработчика событий `Paint` с помощью метода `Graphics.DrawImage()`:

```
public partial class MainForm : Form
{
    private Bitmap[] myImages = new Bitmap[3];
    public MainForm()
    {
        // Загрузить некоторые локальные изображения.
        myImages[0] = new Bitmap("imageA.bmp");
        myImages[1] = new Bitmap("imageB.bmp");
        myImages[2] = new Bitmap("imageC.bmp");
        CenterToScreen();
        InitializeComponent();
    }
```

```
    private void MainForm_Paint(object sender, PaintEventArgs e)
    {
        Graphics g = e.Graphics;
        // Визуализировать все три изображения.
        int yOffset = 10;
        foreach (Bitmap b in myImages)
        {
            g.DrawImage(b, 10, yOffset, 90, 90);
            yOffset += 100;
        }
    }
}
```

На заметку! Файлы `*.bmp`, применяемые в данном примере, должны находиться в той же папке, что и приложение (или специфицироваться жестко закодированными путями). Позднее в этой главе мы преодолеем это ограничение.

На рис. 3.18 показан вывод.

Рис. 3.18. Визуализация изображений

И, наконец, имейте в виду, что, невзирая на название, класс `Bitmap` может содержать графические данные, хранящиеся в любых графических форматах (`*.tif`, `*.gif`, `*.bmp` и т.п.).

Исходный код. Приложение `BasicImages` включено в подкаталог `Bonus Chapter 3`.

Перетаскивание и проверка попадания в элемент управления `PictureBox`

Хотя вы вольны визуализировать образы `Bitmap` непосредственно на поверхности любого типа-наследника `Control`, максимальный контроль и объем функциональности дает применение для вывода изображений типа `PictureBox`. Например, поскольку `PictureBox` “является” `Control`, от него наследуется большой объем функциональности, включая способность обрабатывать различные события, присваивать всплывающую подсказку или контекстное меню, и т.д. Хотя того же можно достичь с помощью `Bitmap`, при этом придется написать значительный объем рутинного кода.

Чтобы доказать полезность типа PictureBox, давайте создадим простую "игру", иллюстрирующую способность фиксировать действия мыши над графическим изображением. Если пользователь щелкнет кнопкой мыши где-то в пределах границ изображения, он переключается в режим "перетаскивания" и сможет перемещать это изображение по всей поверхности Form. Чтобы сделать задачу еще интереснее, давайте будем также отслеживать, где пользователь отпустит изображение. Если это окажется в пределах визуализированного с помощью GDI+ прямоугольника, будут предприняты некоторые дополнительные действия (о которых чуть позже). Как должно быть известно, процесс проверки события щелчка кнопкой мыши внутри определенной области называется проверкой попадания (hit testing).

Тип PictureBox получает большую часть своей функциональности от базового класса Control. Ранее вы уже познакомились с множеством членов Control, поэтому давайте переключим внимание на процесс присваивания изображения переменной-члену PictureBox свойства Image (файл happyDude.bmp должен находиться в каталоге приложения):

```
public partial class MainForm : Form
{
    // Член хранит изображения улыбающейся рожицы.
    private PictureBox happyBox = new PictureBox();
    public MainForm()
    {
      // Конфигурируем PictureBox.
      happyBox.SizeMode = PictureBoxSizeMode.StretchImage;
      happyBox.Location = new System.Drawing.Point(64, 32);
      happyBox.Size = new System.Drawing.Size(50, 50);
      happyBox.Cursor = Cursors.Hand;
      happyBox.Image = new Bitmap("happyDude.bmp");
      // Добавляем в коллекции Controls формы.
      Controls.Add(happyBox);
    }
}
```

Помимо Image, единственное свойство, представляющее для нас интерес — это SizeMode, которое использует перечисление PictureBoxSizeMode. Этот тип служит для управления тем, как ассоциированное изображение должно быть визуализировано внутри ограничивающего прямоугольника PictureBox. Здесь вы присваиваете PictureBoxSizeMode.StretchImage, что говорит о том, что требуется "натянуть" графическое изображение на всю область элемента PictureBox (установленную с размером 50 50 пикселей).

Следующая задача — обработать события MouseMove, MouseUp и MouseDown элемента PictureBox, используя синтаксис событий C#:

```
public MainForm()
{
...
    // Добавить обработчики для следующих событий.
    happyBox.MouseDown += new MouseEventHandler(happyBox_MouseDown);
    happyBox.MouseUp += new MouseEventHandler(happyBox_MouseUp);
    happyBox.MouseMove += new MouseEventHandler(happyBox_MouseMove);
    Controls.Add(happyBox);
    InitializeComponent();
}
```

Обработчик MouseDown отвечает за сохранение входящих координат (*x*, *y*) курсора внутри двух переменных-членов System.Int32 (oldX и oldY) для последующего использования, а также за установку переменной-члена System.Boolean (isDragging) в

true, чтобы показать, что операция перетаскивания находится в процессе выполнения. Добавьте следующие переменные-члены в класс Form и реализуйте обработчик MouseDown следующим образом:

```
private void happyBox_MouseDown(object sender, MouseEventArgs e)
{
   isDragging = true;
   oldX = e.X;
   oldY = e.Y;
}
```

Обработчик события MouseMove просто перемещает положение PictureBox (с применением свойств Top и Left), сравнивая текущее положение курсора со значением, полученным во время события MouseDown:

```
private void happyBox_MouseMove(object sender, MouseEventArgs e)
{
   if (isDragging)
   {
      // Необходимо определить новое значение Y на основе
      // того, где произошел щелчок кнопкой мыши.
      happyBox.Top = happyBox.Top + (e.Y - oldY);
      // То же самое для X (использовать oldX как базовую величину).
      happyBox.Left = happyBox.Left + (e.X - oldX);
   }
}
```

Обработчик событий MouseUp устанавливает булевскую переменную isDragging в false, чтобы обозначить конец операции перетаскивания. К тому же, если событие MouseUp случается, когда PictureBox находится внутри визуализированного GDI+ образа Rectangle, можно предположить, что пользователь выиграл игру (хоть и довольно примитивную). Добавьте в класс Form переменную-член Rectangle (по имени dropRect), установленную в заданный размер:

```
public partial class MainForm : Form
{
   private PictureBox happyBox = new PictureBox();
   private int oldX, oldY;
   private bool isDragging;
   private Rectangle dropRect = new Rectangle(100, 100, 140, 170);
...
}
```

Обработчик события MouseUp теперь может быть реализован следующим образом:

```
private void happyBox_MouseUp(object sender, MouseEventArgs e)
{
  isDragging = false;
  // Курсор мыши внутри области прямоугольника сброса?
  if(dropRect.Contains(happyBox.Bounds))
     MessageBox.Show("You win!", "What an amazing test of skill...");
}
```

И, наконец, необходимо визуализировать прямоугольную область (поддерживаемую переменной-членом dropRect) в Form внутри обработчика событий Paint:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   // Нарисовать прямоугольник сброса.
   Graphics g = e.Graphics;
   g.FillRectangle(Brushes.AntiqueWhite, dropRect);
```

```
    // Показать инструкцию.
    g.DrawString("Drag the happy guy in here...",
    new Font("Times New Roman", 25), Brushes.Red, dropRect);
}
```

После запуска приложения, вы увидите то, что показано на рис. 3.19.

Рис. 3.19. Простая игра

Если вы умудритесь выиграть в эту игру, то получите в награду грамоту, показанную на рис. 3.20.

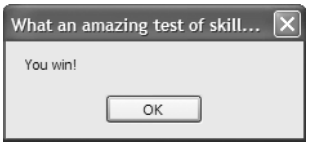

Рис. 3.20. Вы победили!

Исходный код. Приложение `DraggingImages` включено в подкаталог `Bonus Chapter 3`.

Проверка попаданий в визуализированные изображения

Проверка попадания курсора мыши в тип-наследник `Control` (такой как `PictureBox`) очень проста, поскольку он может непосредственно реагировать на события мыши. Однако что если вы хотите проверить попадание в геометрическую фигуру, визуализированную непосредственно на поверхности `Form`?

Чтобы проиллюстрировать процесс, давайте вернемся к предыдущему приложению `BasicImages` и добавим новую функциональность. Целью будет определение, на каком из трех изображений пользователь выполнил щелчок. После обнаружения такого изображения изменяется свойство `Text` объекта `Form`, а изображение подсвечивается 5-пиксельной рамкой.

Первый шаг состоит в определении нового набора переменных-членов в типе `Form`, представляющих объекты `Rectangle`, попадание в которые будет проверяться в событии `MouseDown`. Когда событие произойдет, нужно программно определить, находятся ли

координаты курсора мыши (*x*, *y*) где-то в границах прямоугольников `Rectangle`, представляющих измерения каждого `Image`. Если пользователь щелкнул на определенном изображении, приватная булевская переменная (`isImageClicked`) должна быть установлена в `true`. Затем с помощью другой переменной-члена типа перечисления по имени `ClickedImage` (показано ниже) определяется, какое именно изображение было выбрано:

```
enum ClickedImage
{
   ImageA, ImageB, ImageC
}
```

С учетом всего этого, начальная модификация класса-наследника `Form` будет такой:

```
public partial class MainForm : Form
{
   private Bitmap[] myImages = new Bitmap[3];
   private Rectangle[] imageRects = new Rectangle[3];
   private bool isImageClicked = false;
   ClickedImage imageClicked = ClickedImage.ImageA;
   public MainForm()
   {
   ...
      // Установить прямоугольники.
      imageRects[0] = new Rectangle(10, 10, 90, 90);
      imageRects[1] = new Rectangle(10, 110, 90, 90);
      imageRects[2] = new Rectangle(10, 210, 90, 90);
   }
   private void MainForm_MouseDown(object sender, MouseEventArgs e)
   {
      // Получить координаты (x, y) щелчка мыши.
      Point mousePt = new Point(e.X, e.Y);
      // Проверить попадание курсора мыши в один из трех Rectangle.
      if (imageRects[0].Contains(mousePt))
      {
         isImageClicked = true;
         imageClicked = ClickedImage.ImageA;
         this.Text = "You clicked image A";
      }
      else if (imageRects[1].Contains(mousePt))
      {
         isImageClicked = true;
         imageClicked = ClickedImage.ImageB;
         this.Text = "You clicked image B";
      }
      else if (imageRects[2].Contains(mousePt))
      {
         isImageClicked = true;
         imageClicked = ClickedImage.ImageC;
         this.Text = "You clicked image C";
      }
      else // Ни в один не попали, установить значения по умолчанию.
      {
         isImageClicked = false;
         this.Text = "Hit Testing Images";
      }
      // Перерисовать клиентскую область.
      Invalidate();
   }
}
```

Обратите внимание, что финальная проверка условий устанавливает переменную-член `isImageClicked` в `false`, что указывает на то, что пользователь не выбрал ни одно из трех изображений. Это важно, поскольку нужно удалить выделение предыдущего выбранного изображения. Как только элементы проверены, клиентская область объявляется недостоверной. Ниже показан модифицированный обработчик события `Paint`:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   // Визуализировать три изображения.
...
   // Нарисовать рамку (если был выполнен щелчок).
   if (isImageClicked == true)
   {
      Pen outline = new Pen(Color.Tomato, 5);
      switch (imageClicked)
      {
         case ClickedImage.ImageA:
            g.DrawRectangle(outline, imageRects[0]);
            break;
         case ClickedImage.ImageB:
            g.DrawRectangle(outline, imageRects[1]);
            break;
         case ClickedImage.ImageC:
            g.DrawRectangle(outline, imageRects[2]);
            break;
         default:
            break;
      }
   }
}
```

Теперь можно запустить приложение и удостовериться, что подсвеченная рамка появляется вокруг каждого изображения, на котором выполняется щелчок (и пропадает после щелчка вне границ изображений).

Проверка попаданий в непрямоугольные изображения

Теперь пусть необходимо выполнить проверку попадания в непрямоугольную область. Предположим, что приложение изменено так, чтобы визуализировалась неправильные геометрические фигуры, которые также поддерживают выделение щелчком кнопкой мыши (рис. 3.21).

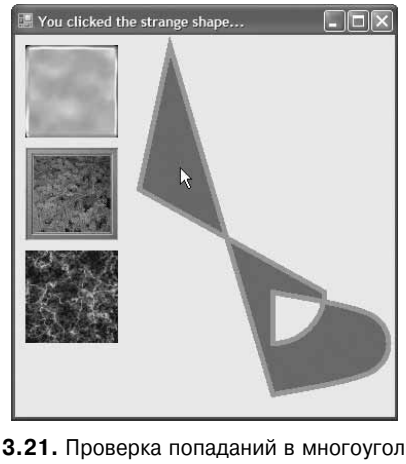

Рис. 3.21. Проверка попаданий в многоугольники

Это геометрическое изображение визуализировалось в `Form` с использованием метода `FillPath()` типа `Graphics`. Этот метод принимает экземпляр объекта `GraphicsPath`, который инкапсулирует серии связанных отрезков прямых, кривых и строк. Добавление новых элементов к экземпляру `GraphicsPath` обеспечивается несколькими последовательными вызовами методов `Add`*XXX*`()`, описанных в табл. 3.9.

Таблица 3.9. Методы добавления элементов к классу `GraphicsPath`

Методы	Назначение
`AddArc()`	Добавляет эллиптическую дугу к текущей фигуре.
`AddBezier()` `AddBeziers()`	Добавляет кубическую кривую Безье (иди множество кривых Безье) к текущей фигуре.
`AddClosedCurve()`	Добавляет замкнутую кривую к текущей фигуре.
`AddCurve()`	Добавляет кривую к текущей фигуре.
`AddEllipse()`	Добавляет эллипс к текущей фигуре.
`AddLine()` `AddLines()`	Добавляет линейный сегмент к текущей фигуре.
`AddPath()`	Добавляет специфицированный `GraphicsPath` к текущей фигуре.
`AddPie()`	Добавляет сегмент круга к текущей фигуре.
`AddPolygon()`	Добавляет многоугольник к текущей фигуре.
`AddRectangle()` `AddRectangles()`	Добавляет один (или более) прямоугольник к текущей фигуре.
`AddString()`	Добавляет текстовую строку к текущей фигуре.

Специфицируйте ссылку `using` для пространства имен `System.Drawing.Drawing2D` и добавьте новую переменную-член `GraphicsPath` в класс-наследник `Form`. В конструкторе формы постройте набор элементов, представляющих путь следующим образом:

```
public partial class MainForm : Form
{
   GraphicsPath myPath = new GraphicsPath();
...
   public MainForm()
   {
      // Создать интересный путь.
      myPath.StartFigure();
      myPath.AddLine(new Point(150, 10), new Point(120, 150));
      myPath.AddArc(200, 200, 100, 100, 0, 90);
      Point point1 = new Point(250, 250);
      Point point2 = new Point(350, 275);
      Point point3 = new Point(350, 325);
      Point point4 = new Point(250, 350);
      Point[] points = { point1, point2, point3, point4} ;
      myPath.AddCurve(points);
      myPath.CloseFigure();
   ...
   }
}
```

Обратите внимание на вызовы `StartFigure()` и `CloseFigure()`. При вызове `StartFigure()` можно вставить новый элемент в путь, который строится.

Вызов `CloseFigure()` замыкает текущую фигуру и начинает новую (если это нужно). Также знайте, что если фигура содержит последовательность соединенных прямых и кривых линий (как в случае экземпляра `myPath`), петля замыкается соединением линии от конечной точки к начальной. Добавьте дополнительное имя `StrangePath` к перечислению `ImageClicked`:

```
enum ClickedImage
{
   ImageA, ImageB,
   ImageC, StrangePath
}
```

Теперь модифицируйте существующий обработчик событий `MouseDown`, добавив проверку попадания позиции курсора (*x*, *y*) в границы `GraphicsPath`. Подобно типу `Region`, это можно исследовать с помощью члена `IsVisible()`:

```
protected void OnMouseDown (object sender, MouseEventArgs e)
{
   // Получить координаты (x, y) щелчка мыши.
   Point mousePt = new Point(e.X, e.Y);
...
   else if(myPath.IsVisible(mousePt))
   {
      isImageClicked = true;
      imageClicked = ClickedImage.StrangePath;
      this.Text = "You clicked the strange shape...";
   }
...
}
```

И, наконец, обновите обработчик `Paint` следующим образом:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
...
   // Рисовать графический путь.
   g.FillPath(Brushes.Sienna, myPath);

   // Рисовать рамку (если был совершен щелчок).
   if(isImageClicked == true)
   {
      Pen outline = new Pen(Color.Red, 5);
      switch(imageClicked)
      {
      ...
         case ClickedImage.StrangePath:
            g.DrawPath(outline, myPath);
            break;
         default:
            break;
      }
   }
}
```

Исходный код. Проект `HitTestingImages` включен в подкаталог `Bonus Chapter 3`.

Формат ресурсов .NET

Вплоть до этого места главы каждое созданное приложение использовало внешние ресурсы (такие как файлы битовых карт), требуя, чтобы файлы графических изображений находились внутри каталога клиентского приложения. Учитывая это, файлы *.bmp загружались по абсолютным именам:

```
// Заполнить образы из битовых карт.
bMapImageA = new Bitmap("imageA.bmp");
bMapImageB = new Bitmap("imageB.bmp");
bMapImageC = new Bitmap("imageC.bmp");
```

Эта логика, конечно, требовала наличия в каталоге приложения этих трех файлов imageA.bmp, imageB.bmp и imageC.bmp; в противном случае возникало исключение времени выполнения.

Вспомните, что сборка — это коллекция типов и *дополнительных ресурсов*. Учитывая это, последняя задача в этой главе состоит в том, чтобы научиться встраивать внешние ресурсы (наподобие графических файлов и строк) в саму сборку. Таким образом, двоичный файл становится по-настоящему самодостаточным. На самом низком уровне встраивание внешних ресурсов в сборку .NET включает следующие шаги.

1. Создать файл *.resx, содержащий пары имя/значение для каждого ресурса приложения через представление данных XML.
2. Использовать утилиту командной строки resgen.exe для преобразования XML-файла *.resx в двоичный эквивалент (файл *.resource).
3. Используя флаг /resource компилятора C#, встроить двоичный файл *.resources в сборку.

Как и можно было ожидать, эти шаги автоматизированы в среде Visual Studio. Очень скоро вы узнаете, как данная интегрированная среда разработки может помочь в этом. А пока давайте посмотрим, как сгенерировать и встроить ресурсы .NET в командной строке.

Пространство имен System.Resources

Ключ к пониманию формата ресурсов .NET состоит в знании типов, определенных в пространстве имен System.Resources. Этот набор типов предоставляет средства для чтения и записи файлов *.resx (на базе XML) и файлов *.resources (двоичных), а также получения ресурсов, встроенных в конкретную сборку. В табл. 3.10 приведен краткий список основных типов.

Таблица 3.10. Члены пространства имен System.Resources

Члены	Назначение
ResourceReader ResourceWriter	Эти типы позволяют читать и записывать двоичные файлы *.resources.
ResXResourceReader ResXResourceWriter	Эти типы позволяют читать и записывать XML-файлы *.resx.
ResourceManager	Этот тип позволяет программно получать встроенные ресурсы из определенной сборки.

Программное создание файла *.resx

Как уже упоминалось, файл *.resx представляет собой блок данных XML, который присваивает пары имя/значение каждому ресурсу приложения.

Класс ResXResourceWriter представляет набор членов, которые позволяют создавать файл *.resx, добавлять к нему двоичные и строковые ресурсы, а также фиксировать их в хранилище. Для иллюстрации давайте создадим простое приложение (ResXWriter), которое сгенерирует файл *.resx, содержащий вхождение для файла happyDude.bmp (который мы использовали в примере DraggingImages) и одиночный строковый ресурс. Графический интерфейс будет состоять из единственного элемента Button (рис. 3.22).

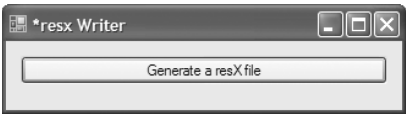

Рис. 3.22. Приложение ResXWriter

Обработчик события Click для Button добавит happyDude.bmp и строковый ресурс к файлу *.resx, который будет сохранен на локальном диске C:

```
private void btnGenResX_Click(object sender, EventArgs e)
{
    // Создать писатель resx и указать файл для записи.
    ResXResourceWriter w = new ResXResourceWriter(@"C:\ResXForm.resx");
    // Добавить картинку и строку.
    Image i = new Bitmap("happyDude.bmp");
    w.AddResource("happyDude", i);
    w.AddResource("welcomeString", "Hello new resource format!");
    // Зафиксировать файл.
    w.Generate();
    w.Close();
}
```

Член, представляющий интерес — ResXResourceWriter.AddResource(). Этот метод перегружен несколько раз, чтобы позволить вставлять двоичные данные (как это делалось с файлом happyDude.bmp), а также текстовые данные (как было сделано с текстовой строкой). Обратите внимание, что каждая версия принимает два параметра: имя заданного ресурса в файле *.resx и сами данные. Метод Generate() сбрасывает информацию в файл. В этот момент вы получаете XML-описание ресурсов — графического образа и строки. Чтобы удостовериться в этом, откройте файл ResXForm.resx в текстовом редакторе (рис. 3.23).

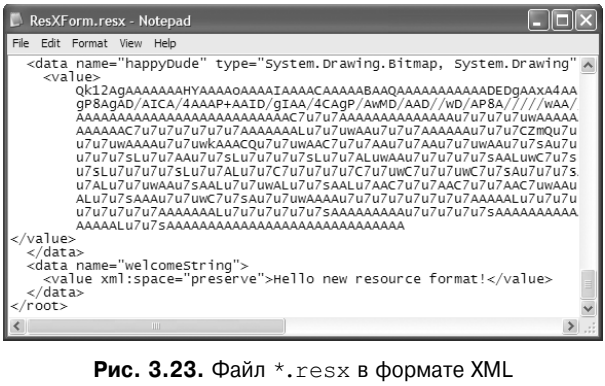

Рис. 3.23. Файл *.resx в формате XML

Построение файла *.resources

Имея файл *.resx, с помощью утилиты resgen.exe можно сгенерировать его двоичный эквивалент. Для этого откройте окно командной строки Visual Studio, перейдите на диск C и введите следующую команду:

```
resgen resxform.resx resxform.resources
```

После этого можно открыть файл *.resources в Visual Studio (рис. 3.24).

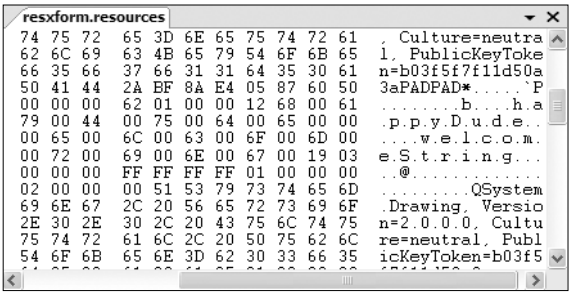

Рис. 3.24. Двоичный файл *.resources

Встраивание файла *.resources в сборку .NET

Теперь вы готовы к тому, чтобы встроить файл *.resources в сборку .NET, используя аргумент командной строки /resources компилятора C#. Для иллюстрации скопируйте файлы Program.cs, Form1.cs и Form1.Designer.cs на диск C, откройте окно командной строки Visual Studio и введите следующую команду:

```
csc /resource:resxform.resources /r:System.Drawing.dll *.cs
```

Теперь, открыв сборку с помощью ildasm.exe, вы найдете там обновленный манифест, как показано на рис. 3.25.

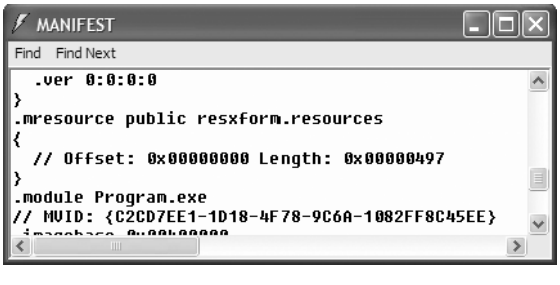

Рис. 3.25. Встроенные ресурсы

Работа с ResourceWriter

В предыдущем примере использовались типы ResXResourceWriter для генерации файла XML, содержащего пары имя/значение для каждого ресурса приложения. Результирующий файл *.resx затем прогонялся через утилиту resgen.exe. И, наконец, файл *.resources встраивался в сборку с помощью флага /resources компилятора C#. По правде говоря, строить файл *.resx было не обязательно (хотя читабельное XML-представление ресурсов вполне может пригодиться). Если файл *.resx не нужен,

с помощью типа ResourceWriter можно непосредственно создать двоичный файл *.resources:

```
private void GenerateResourceFile()
{
    // Создать новый файл *.resources.
    ResourceWriter rw;
    rw = new ResourceWriter(@"C:\myResources.resources");
    // Добавить одну картинку и одну строку.
    rw.AddResource("happyDude", new Bitmap("happyDude.bmp"));
    rw.AddResource("welcomeString", "Hello new resource format!");
    rw.Generate();
    rw.Close();
}
```

После этого полученный файл *.resources можно встроить в сборку, указав флаг компилятора /resources:

```
csc /resource:myresources.resources *.cs
```

Исходный код. Проект ResXWriter включен в подкаталог Bonus Chapter 3.

Генерация ресурсов с использованием Visual Studio

Хотя вполне возможно работать с файлами *.resx/*.resources вручную в командной строке, знайте, что Visual Studio автоматизирует создание и встраивание ресурсов в проекты. Для иллюстрации создайте новое приложение Windows Forms по имени MyResourcesWinApp. Теперь, открыв Solution Explorer, вы обнаружите, что каждая форма в приложении автоматически сопровождается ассоциированным файлом *.resx (рис. 3.26).

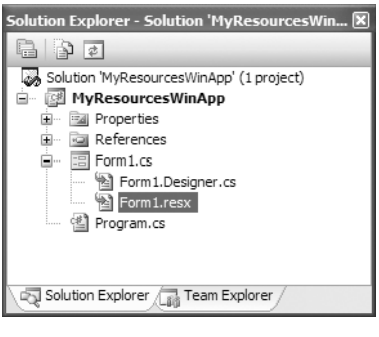

Рис. 3.26. Автоматически сгенерированные файлы *.resx в Visual Studio

Этот файл *.resx будет поддерживаться автоматически, пока вы на самом деле не добавите ресурсы (вроде картинки для виджета PictureBox) посредством визуальных дизайнеров. Обновлять этот файл вручную, чтобы специфицировать специальные ресурсы, *не* придется, поскольку Visual Studio автоматически регенерирует этот файл при каждой компиляции. Поэтому следует позволить IDE-среде поддерживать файлы *.resx форм самостоятельно.

Когда необходимо поддерживать специальный набор ресурсов, которые не отображаются напрямую на определенную форму, просто вставьте в проект новый файл *.resx (в данном примере — MyCustomResources.resx), используя пункт меню Project⇨Add New Item (рис. 3.27).

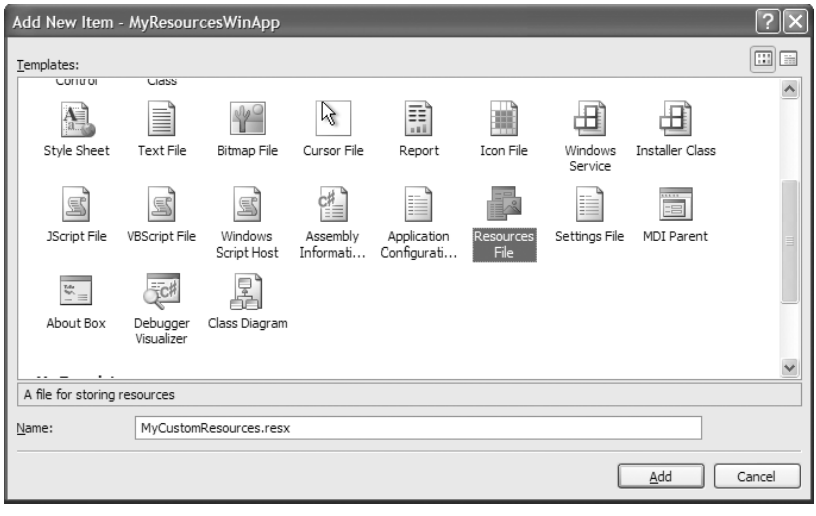

Рис. 3.27. Вставка нового файла `*.resx`

Если открыть новый файл `*.resx`, появится дружественный редактор, которые позволит вставлять строковые данные, файлы изображений, звуковые клипы и прочие ресурсы. Самое левое раскрывающееся меню позволяет указать тип добавляемого ресурса. Сначала добавьте новый строковый ресурс по имени `WellcomeString`, содержащий нужное сообщение (рис. 3.28).

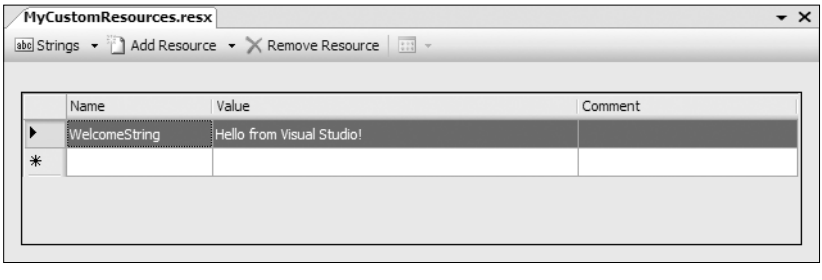

Рис. 3.28. Вставка новых строковых ресурсов в редакторе `*.resx`

Затем добавьте файл изображения `happyDude.bmp`, выбрав пункт Images (Изображения) в левом раскрывающегося меню, затем опцию Add Existing File (рис. 3.29) и перейдя к файлу `happyDude.bmp`.

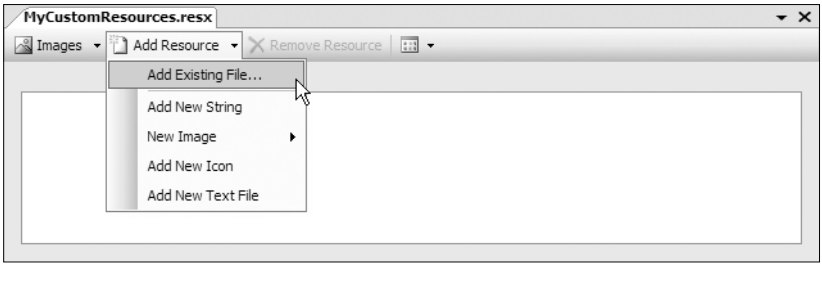

Рис. 3.29. Вставка ресурсов `*.bmp` в редакторе `*.resx`

После этого вы обнаружите, что файл *.bmp был скопирован в каталог приложения. Если выбрать пиктограмму happyDude в редакторе *.resx, можно указать, что изображение должно быть включено непосредственно в сборку (вместо привязки в качестве отдельного внешнего файла), изменив свойство Persistence (рис. 3.30).

Рис. 3.30. Встраивание специфицированных ресурсов

Вдобавок Solution Explorer теперь показывает новую папку под названием Resources, которая содержит каждый элемент, встраиваемый в сборку. Как и можно было догадаться, при открытии определенного ресурса Visual Studio запустит соответствующий ассоциированный редактор. Если теперь скомпилировать приложение, данные строки и изображения будут встроены в сборку.

Программное чтение ресурсов

После того, как вы ознакомились с процессом встраивания ресурсов в сборку (используя csc.exe или Visual Studio), следует изучить способ программного их чтения для использования в приложении посредством типа ResourceManager. Для иллюстрации добавьте виджеты Button и PictureBox к типу Form (рис. 3.31).

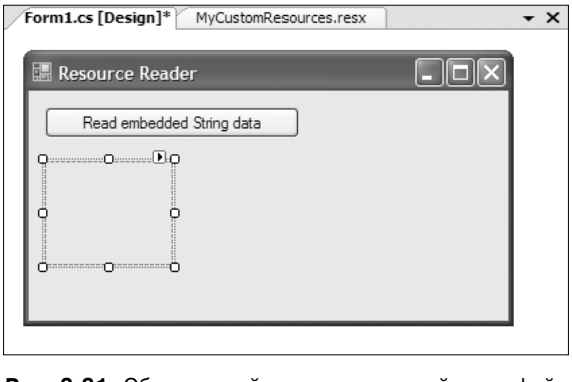

Рис. 3.31. Обновленный пользовательский интерфейс

Затем обработайте событие Click нового элемента Button. Модифицируйте обработчик события следующим образом:

```
// Не забудьте добавить using для System.Resources и System.Reflection!
private void btnGetStringData_Click(object sender, EventArgs e)
{
    // Создать диспетчер ресурсов.
    ResourceManager rm =
        new ResourceManager("MyResourcesWinApp.MyCustomResources",
        Assembly.GetExecutingAssembly());

    // Получить встроенную строку (зависит от регистра).
    MessageBox.Show(rm.GetString("WelcomeString"));

    // Получить встроенное битовое изображение (зависит от регистра).
    myPictureBox.Image = (Bitmap)rm.GetObject("HappyDude");

    // Очистить.
    rm.ReleaseAllResources();
}
```

Обратите внимание, что первый аргумент конструктора `ResourceManager` — полностью квалифицированное имя файла `*.resx` (без расширения). Второй параметр — ссылка на сборку, содержащую встроенный ресурс (в данном случае — текущая сборка). После создания экземпляра `ResourceManager` можно вызывать `GetString()` или `GetObject()` для извлечения встроенных данных. Запустив приложение и щелкнув на кнопке, вы обнаружите, что данные строки отображаются в `MessageBox`, а изображение извлекается из сборки и помещается в `PictureBox`.

Исходный код. Проект `MyResourcesWinApp` включен в подкаталог `Bonus Chapter 3`.

Итак, на этом знакомство с GDI+ и пространствами имен `System.Drawing` завершено. Если вы заинтересованы в углубленном изучении GDI+ (включая поддержку печати), обращайтесь к изданию *GDI+ Programming in C# and VB .NET* by Nick Symmonds (Apress, 2002 г.).

Резюме

GDI+ — это общее название для множества взаимосвязанных пространств имен .NET, каждое из которых используется для визуализации графических изображений в объектах типов-наследников `Control`. Большая часть этой главы была посвящена изучению работы таких основных типов объектов GDI+, как цвета, шрифты, графические изображения, перья и кисти, в сочетании с мощным типом `Graphics`. Попутно вы ознакомились с некоторыми деталями работы с GDI+, вроде контроля попадания и перетаскивания изображений.

Эта глава завершилась изучением нового формата ресурсов .NET. Как было показано, `*.resx` описывает ресурсы, заданные парами имя/значение, в формате XML. Этот файл может быть передан утилите `resgen.exe`, которая генерирует из него файл двоичного формата (`*.resources`), который можно встраивать в связанную сборку. И, наконец, тип `ResourceManager` предоставляет простой способ для программного извлечения ресурса во время выполнения.