コンポーネント化のメリットと手法を追求する

Visual Studio .NETで究めるコンポーネント開発

最終回

コントロール実行時の動作を制御する

株式会社コムラッド
辻 慶 *TSUJI, Kei*
http://www.comrade.co.jp/

Technology Tools

- ❑ Visual Basic .NET
- ☑ Visual C# .NET
- ❑ SQL Server 2000
- ❑ Oracle 9i
- ❑ Access 2002
- ❑ ASP.NET
- ❑ Internet Information Services
- ❑ Other:

Level

★ ★ ★ ★ ★

Samples

・本稿で取り上げたソースコードおよびサンプルプログラムは、付録CD-ROMの¥DOTNET¥NETARCH02ディレクトリに収録しています。

¥DRAW
描画関連のサンプル

¥SCROLL
スクロール関連のサンプル

¥CONTROLS
記事末コラムのサンプル

はじめに

本連載では、これまでデザイン時の挙動に関して重点的に説明をしてきましたが、今回は、実行時の挙動に関連するテクニックについてご紹介します。

カスタムペイント
サンプル：NormalDraw

これまでカスタムペイント（独自のUI）のコントロールの作り方に関して触れていなかったので、ここで簡単に説明しておきましょう。

描画はPaintイベントが発生したタイミングで行ないます。方法はたいていのイベントと同様に、イベントハンドラを登録する方法と、protectedなイベントハンドラ用のメソッドをオーバーライドする方法があります。

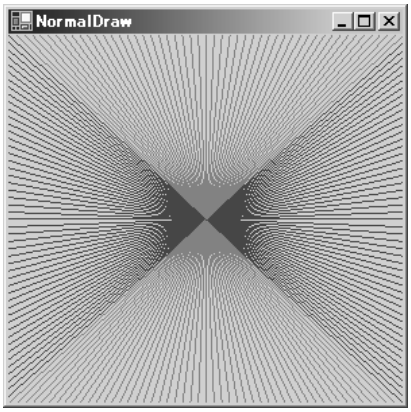
イベントハンドラを登録する方法をリスト1に示します。フォームに対する描画もコントロールに対する描画も基本は同じなので、サンプルではフォームを使って記述しています。

また、ここではDrawAsterメソッドにPaintEventArgsのGraphicsプロパティ（Graphicsオブジェクト）を渡すことで実際の描画を行なっています。描画結果は、図1のように、星のような線がフォームのクライアント領域に描かれます。

全体を再描画する
サンプル：ClipArea/ResizeRedraw

Graphicsクラスは「GDI＋」を使用した描画の要となるクラスです。ここで、Graphicsクラスの一部の機能に関して簡単に解説しておきます（誌面の

図1：リスト1の描画結果



リスト1：イベントハンドラを使用したカスタムペイント

```
private void InitializeComponent() {
    （略）
   Paint += new System.Windows.Forms.PaintEventHandler(
         this.NormalDraw_Paint);
}

const float LINEFREQ = 50F;
private void DrawAster(Graphics g) {
   float width = ClientRectangle.Width;
   float height = ClientRectangle.Height;
   float x_delta = width / LINEFREQ;
   float y_delta = height / LINEFREQ;
   for (int i = 0; i < LINEFREQ; i++) {
      g.DrawLine(blue, 0, y_delta * i,
            width, height - (y_delta * i));
      g.DrawLine(red, x_delta * (i + 1), 0,
            width - (x_delta * (i + 1)), height);
   }
}

private void NormalDraw_Paint(object sender, PaintEventArgs e) {
   DrawAster(e.Graphics);
}
```

都合上詳細な解説は行ないませんので、リファレンスなどを参照してください)。

Graphicsクラスには、描画に関するさまざまな系統のメソッドやプロパティが存在します。

- **Draw系：**Penクラスを使用して、線を描画する
- **Fill系：**Brushクラスを使用して、面を塗る

また、Clipを変更することで、描画の領域を制限したり、SmoothigModeプロパティで、アンチエイリアスの設定を行なったりすることもできます（リスト2)。

先ほどのサンプルNormalDrawを実際に実行して、フォームのサイズを変更してみてください。拡大した場合が図2、縮小した場合が図3のようになります。このように、標準の描画は更新する必要があるところに対してのみ行なわれます。

描画される領域に関しては、PaintEventArgsのClipRectangleで取得可能です。

リスト3のClipArea_Paintメソッドのようなイベントハンドラをフォームに登録することで、この挙動を正確にモニターすることができます（サンプルClipArea)。ここでは、念のためにOnPaintメソッドをオーバーライドしてみましたが、特に違いはないようです。これにより、拡大時は新たに露出した部分だけ描画され、縮小時はPaintイベントがまったく発生しないことがわかります。

リスト2：アンチエイリアスの設定

```
private void Smooth_Paint(object sender, PaintEventArgs e) {
   e.Graphics.SmoothingMode = SmoothingMode.AntiAlias;
   // 以下に実際の描画ルーチン
}
```

リスト3：Paintイベントハンドラでのクリップ領域の出力

```
private void ClipArea_Paint(object sender, PaintEventArgs e) {
   Console.WriteLine("PaintEvent : {0}", e.ClipRectangle);
}
protected override void OnPaint(PaintEventArgs e) {
   Console.WriteLine("OnPaint : {0}", e.ClipRectangle);
   base.OnPaint(e);
}
```

図2：サンプルNormalDrawを描画結果を拡大した場合

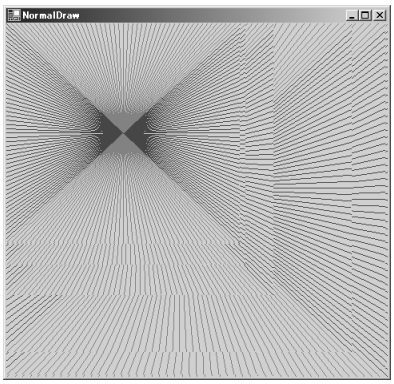
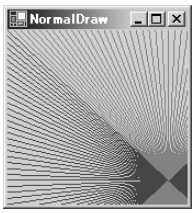


図3：サンプルNormalDrawを描画結果を縮小した場合



この仕組みは、再描画する部分を必要最低限に抑えて、描画スピードを上げるのが目的ですが、今回の描画のように、表示されている画面の比率により、「描画が異なる」「伸縮表示の画像を貼る」などの場合、全体に対して再描画したいというニーズもあるでしょう。

この動作を変更するためには、Controlクラスのprotectedメソッドである、SetStyleメソッドを使用して、以下のように設定を変更します。

```
SetStyle(ControlStyles.ResizeRedraw, true);
```

この変更は、InitializeComponentの中で行なってもよいのですが、Visual Studio .NET 2003では、「メソッドSystem.Windows.Forms.Form.SetStyleが見つかりません」というエラーが表示されてしまいますので（SetStyleがprotectedなメ