

29

GDI+ zum Zeichnen von Formular- und Steuerelementinhalten verwenden

-
- 864** Einführung in GDI+
 - 878** Flimmerfreie, fehlerfreie und schnelle Darstellungen von GDI+-Zeichnungen
 - 884** Was Sie beim Zeichnen von breiten Linienzügen beachten sollten
-

GDI+ ist eine Klassenbibliothek, die das Zeichnen verschiedener Elemente in Windows (Windows im Sinne vom Windows-Betriebssystem) erlaubt. GDI steht als Abkürzung von *Graphic Device Interface* – etwa *grafische Geräteschnittstelle* – und das Pluszeichen hinter dem Akronym lässt schon vermuten, dass es sich um etwas Weiterentwickeltes handeln muss.

Allerdings ist das im Grunde genommen nur halb richtig – jedenfalls in der gegenwärtigen Version des GDI+. GDI+ ist der von Microsoft erklärte Nachfolger des GDI, der, wenn auch in ständig weiterentwickelter Form, schon zu 16-Bit-Windows-Zeiten sozusagen den ausführenden Produzenten für alles darstellte, was in irgendeiner Form auf dem Bildschirm erscheinen sollte. Aus diesem Grund erfuhren viele Grundfunktionen des GDI eine umfangreiche Unterstützung durch die Treiber von Grafikkarten der verschiedensten Hersteller. Das heißt im Klartext: Wenn Sie eine Linie von einem zum anderen Punkt auf dem Bildschirm mit GDI zeichnen, dann ist es nicht eine bestimmte Prozedur im GDI, die die eigentlichen Punkte setzt, sondern die Grafikkarte selbst, die diese Aufgabe übernimmt. Das GDI teilt dem Treiber lediglich mit, dass es eine Linie gezeichnet haben möchte. Und genau das ist zurzeit noch der Unterschied zum GDI+. Es bietet viel umfassendere und vor allen Dingen auch einfacher zu handhabende Zeichenfunktionen, doch viele davon sind momentan noch nicht hardwareunterstützt. Und das ist der Grund, weswegen Microsoft mit GDI+ grundsätzlich auf dem richtigen Weg ist, es sich für einige wenige Anwendungen unter Windows aus Geschwindigkeitsgründen aber einfach noch nicht hundertprozentig zur Verwendung eignet.

Nun ist das Framework für die Zukunft konzipiert, und in zukünftigen .NET- und Windows-Betriebssystemversionen werden sich definitiv leistungsfähigere Werkzeuge für die Darstellung von Grafik befinden – die *Windows Presentation Foundation* (auch bekannt unter seinem Codenamen »Avalon«) wird es nach Fertigstellung dieses Buches sowohl noch für Windows XP als natürlich auch für den Windows XP-Nachfolger Windows Vista geben. Aus diesem Grund hat es wohl für die Entwickler des Frameworks bislang keinen Sinn ergeben, die ältere Version des GDI (ohne Plus) als Klassenlibrary in .NET zu implementieren.

Außer Acht gelassen, was die wirklichen Gründe für die Entscheidung zu diesem Schritt waren, gilt: Es ist derzeit jedenfalls nicht vorhanden, und es wird aller Wahrscheinlichkeit nach auch niemals implementiert werden. Für den Moment müssen wir für den Inhalt der Dokumente unserer .NET-Programme mit der aktuellen Version des GDI+ vorlieb nehmen.¹

HINWEIS: GDI+ liefert genug Stoff, um ein eigenes Buch damit zu füllen. Charles Petzold hat das mit seiner Core-Reference,² die ein wenig GDI+-lastig geworden ist, eindrucksvoll unter Beweis gestellt. Aus diesem Grund möchte ich mich an dieser Stelle nur auf grundlegendste Erklärungen zum GDI+ beschränken – gerade auf so viel, dass es für andere Projekte ausreicht – um beispielsweise Benutzersteuerelemente mit sinnvollen Inhalten zu füllen. In den nächsten Abschnitten finden Sie deswegen zunächst die wichtigsten Informationen, die Sie als Grundlage benötigen, um die ersten Gehversuche mit dem GDI+ bewältigen zu können, in sehr kompakter Form. Aus Platzgründen möchte ich mich nicht in endlos langen Beschreibungen von einzelnen Funktionen des GDI+ verlieren. Vielmehr möchte ich mich auf die *Anwendung* des GDI+ zur Lösung bestimmter Probleme beim Entwickeln von Komponenten und Anwendungen konzentrieren. Ersatzweise verwenden Sie für Fragen, die die korrekte Anwendung einer Methode oder eines Konstruktors betreffen, die Online-Hilfe von Visual Studio. Sie leistet gerade beim GDI+ als Referenz ausgezeichnete Arbeit. Durch die IntelliSense-Funktion des Editors werden selbst viele Blicke in die Online-Hilfe überflüssig.

Einführung in GDI+

Wie schon an mehreren Stellen in diesem Buch beschrieben, ist alles, was Sie in ein Windows zeichnen, hoch volatil. Es hält genau so lange, wie sie etwas anderes über das Fenster schieben. Dann ist sein Inhalt verschwunden. Das gilt sogar für das Fenster selbst, das den Inhalt darstellt: Wenn Sie ein Fenster in einer bestimmten Größe auf den Bildschirm gebracht haben, seinen Inhalt zeichnen, es anschließend verkleinern und wieder auf seine alte Größe bringen, ist der Ursprungsinhalt ebenfalls verloren.

Das liegt am Prinzip, wie Windows-Inhalte gemanagt werden: Jedes Fenster ist für das Zeichnen seines Inhaltes selbst verantwortlich. Alle auf `Control` basierenden Komponenten (dazu gehören auch Formulare) müssen deswegen ihr `Paint`-Ereignis behandeln oder noch besser, da schneller, die `OnPaint`-Methode ihrer Basisklasse überschreiben. In beiden Fällen wird den Prozeduren das so genannte `Graphics`-Objekt mit dem `PaintEventArgs`-Objekt übergeben, das den Dreh- und Angelpunkt für alle Grafikoperationen bildet.

¹ Natürlich gibt es auch die Möglichkeit, direkte Betriebssystemaufrufe an das herkömmliche GDI aus .NET heraus durchzuführen – doch das sind dann grundsätzlich so genannte *Unsafe Calls* (unsichere Aufrufe). Unsicher deswegen, weil das Programm das behütete, sichere .NET-Zuhause verlässt. Und solche Schritte sind nur in einer voll vertrauenswürdigen Umgebung erlaubt. Das heißt, dass alle Programme, die unsichere Aufrufe durchführen oder sonst irgendeinen unsicheren Code ausführen, mit den Standardsicherheitseinstellungen nur auf dem Computer selbst ausgeführt werden können. Starten Sie ein solches Programm beispielsweise von einer Netzwerk-Ressource, löst es beim Erreichen des unsicheren Codes eine Sicherheitsausnahme aus.

² Ein sehr empfehlenswertes Buch, gerade wenn es um GDI+ und native Textausgabe/-formatierung geht. Englische Ausgabe (.NET 1.0/1.1), ISBN: 0-7356-1799-6. Deutsche Ausgabe: Windows-Programmierung mit Visual Basic NET, ISBN: 3-86063-691-X.

Die Zeichenroutinen, die dieses Graphics-Objekt nun zur Verfügung stellt, beziehen sich auf den so genannten *Client*-Bereich des Fensters. Das ist das Innere des Fensters, also der Bereich, ohne Rahmen, Titel oder Rollbalken.

Sie können ein Graphics-Objekt übrigens nicht selbst direkt durch seinen Konstruktor erstellen; sie können es nur durch bestimmte Funktionen ermitteln – der Graphics-Parameter, der Ihnen durch `OnPaint` mit `PaintEventArgs` geliefert wird, ist nur ein (wenn auch das am häufigsten auftretende) Beispiel dafür.

Sie haben zwar auch die Möglichkeit, das für eine `Control`-Ableitung gültige Graphics-Objekt auch ohne die `OnPaint`-Ereignisparameter zu bekommen, doch ist das Anwenden dieser Vorgehensweise eher selten der Fall. Außerdem führt es oft zu einer falschen Vorgehensweise, wie das folgende Beispiel zeigt:

BEGLEITDATEIEN: Sie finden dieses Projekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap29\Simple-Gdi01\`.

```
Public Class frmMain

    Private Sub btnLinieZeichnen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) _
        Handles btnLinieZeichnen.Click

        Dim g As Graphics

        'Ermittelt das Graphics-Objekt, das zu einem bestimmten
        'Window gehört, dessen Handle (ID) zur Identifizierung dient.
        g = Graphics.FromHwnd(Me.Handle)

        'Schwarze, ein Pixel dünne Linie zeichnen von (0,0) zu (500,500).
        'Koordinaten werden standardmäßig in Pixel angegeben;
        '(0,0) liegt in der linken, oberen Ecke des Client-Bereichs.
        g.DrawLine(New Pen(Color.Black), 0, 0, 500, 500)

    End Sub

End Class
```

Wenn Sie dieses Programm starten, werden Sie nach dem Anklicken der einzigen Schaltfläche zwei Dinge feststellen: a) Das ermittelte Graphics-Objekt bezieht sich tatsächlich nur auf das Fenster, für das es ermittelt wurde. Denn obwohl der Linienpfad die Schaltfläche kreuzt, zeichnet der Befehl die Linie im Bereich der Schaltfläche nicht (siehe Abbildung 29.1). Und b) Wenn Sie irgendetwas über das Formular bewegen (der Windows-Taschenrechner eignet sich für solche Experimente am besten), ist die Linie verschwunden.

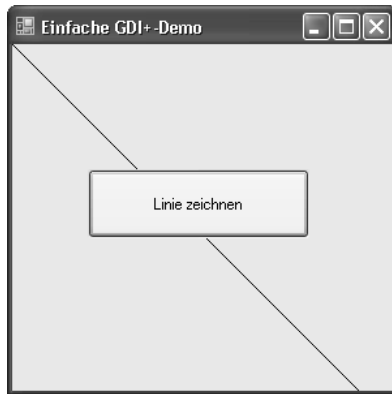


Abbildung 29.1: Zeichenoperationen mit einem Graphics-Objekt beziehen sich nur auf das Fenster, für das das Graphics-Objekt ermittelt wurde

Wenn Sie wollen, dass die Linie auch noch vorhanden ist, *nachdem* sich ein anderes Objekt über dem Formular befunden hat, müssen Sie ein anderes Verfahren verwenden, das grob skizziert folgendermaßen funktioniert:

- Alle Zeichenroutinen finden in der Paint-Ereignisbehandlungsroutine des Formulars statt. Dazu können Sie – wie schon gesagt – entweder das Ereignis im Formular einbinden oder, der zu empfehlende Weg, die Basisprozedur `OnPaint` überschreiben, die das Zeichnen der Linie vornimmt.
- Damit die Linie nur dann gezeichnet wird, wenn die Schaltfläche vom Anwender angeklickt wurde, muss es eine Art Informationsspeicher geben (ein Flag, das von überall im Formular zugänglich ist – also einen Klassen-Member), der `OnPaint` darüber informiert, ob die Linie im Falle eines neu zeichnen Müssens überhaupt gemalt werden soll.
- Damit die Linie auch dann gezeichnet wird, wenn der Anwender die Schaltfläche angeklickt hat, muss der Code zur Behandlung des Klickereignisses für die Schaltfläche nicht nur das Flag setzen, sondern auch den kompletten *Client*-Bereich des Formulars für ungültig erklären. Geschieht das mit einer bestimmten Methode namens `Invalidate`, wird automatisch das Paint-Ereignis ausgelöst, damit `OnPaint` aufgerufen und die Linie gezeichnet werden.

BEGLEITDATEIEN: Sie finden das Projekt, das diese Modifizierungen enthält, im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - SmartClient\Kap29\SimpleGdi02`.

Der dazugehörige Formularcode sieht folgendermaßen aus:

```
Public Class frmMain
    Inherits System.Windows.Forms.Form
    Private myDrawLineFlag As Boolean

    'Klassenweiter Member, der von Click und OnPaint aus zugänglich ist.
    Private myDrawLineFlag As Boolean

    Private Sub btnLinieZeichnen_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles _
        btnLinieZeichnen.Click
```

```

        'Ab sofort darf gezeichnet werden!
        myDrawLineFlag = True
        'Client-Bereich für ungültig erklären --> OnPaint wird ausgelöst,
        'und damit, da myDrawLineFlag jetzt true ist, die Linie gezeichnet.
        Me.Invalidate()

    End Sub

    Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)

        If myDrawLineFlag Then
            Dim g As Graphics = e.Graphics

            'Ermittelt das Graphics-Objekt, das zu einem bestimmten
            'Window gehört, dessen Handle (ID) zur Identifizierung dient.
            g = Graphics.FromHwnd(Me.Handle)

            'Schwarze, ein Pixel dünne Linie zeichnen von (0,0) zu (500,500).
            'Koordinaten werden standardmäßig in Pixel angegeben;
            '(0,0) liegt in der linken, oberen Ecke des Client-Bereichs.
            g.DrawLine(New Pen(Color.Black), 0, 0, 500, 500)
        End If
    End Sub
End Class

```

Wenn Sie diese Version des Programms starten, funktioniert es auf den ersten Blick wie die erste Version des Programms, mit einem entscheidenden Unterschied: Das Formular zeichnet sich in den entscheidenden Schritten neu, und das ist schließlich auch genau das, was es muss.

Sie sollten dieses einfache Prinzip in Ihren eigenen Anwendungen berücksichtigen, wenn Sie selbst gezeichnete Inhalte in Formularen darstellen müssen.

Linien, Flächen, Pens und Brushes

Im Beispiel des letzten Abschnittes haben Sie bereits teilweise sehen können, dass für alle Zeichenoperationen gilt: Sie benötigen entweder ein Pen-Objekt (einen Stift) oder ein Brush-Objekt (einen Pinsel), um Linien oder Flächen zu zeichnen.

Für Linienfiguren benötigen Sie ein Pen-Objekt. Ein Pen bestimmt, wie ein Linienzug gezeichnet werden soll – oder genauer: in welcher Farbe oder welcher Stärke das geschehen soll. Wie die Anwendung des Pen-Objektes generell funktioniert, konnten Sie bereits im vergangenen Beispiel sehen. Der Pen-Konstruktor verfügt über mehrere Überladungen. Neben Farbe und Linienstärke haben Sie die Möglichkeit, ein Pen-Objekt auch aus einem Brush-Objekt zu erstellen und so dickere Linien oder Umrisse bestimmter geometrischer Figuren mit einer spezifischen Füllung zu versehen.

Brush-Objekte in GDI+ sind vielseitig. Sie dienen der Bestimmung der Eigenschaften von Flächenfüllungen. Das Brush-Objekt selbst ist, anders als ein Pen, eine abstrakte Basisklasse, es kann also nicht direkt instanziiert und verwendet werden. Stattdessen gibt es in GDI+ fünf verschiedene Brush-Ableitungen, die unterschiedliche Aufgaben beim Erstellen von Füllfarben bzw. -mustern für Flächen erfüllen:

- **SolidBrush:** Definiert einen simplen, einfarbigen Pinsel. Flächen, die Sie mit einem `SolidBrush` füllen, werden mit der durch den `Brush` bestimmten Farbe ausgefüllt.
- **HatchBrush:** Definiert einen rechteckigen Pinsel mit einer Schraffurart; Vorder- und Hintergrundfarbe der Schraffur sind dabei getrennt einstellbar.
- **TextureBrush:** Definiert einen Pinsel aus einer Bitmap. In welcher Form die Bitmap als Zeichen-vorlage für eine Füllung verwendet wird, stellen Sie über den Konstruktor ein. Interessantes hierzu finden Sie in der VS.NET-Onlinehilfe unter dem `ImageAttributes`-Objekt und der `WrapMode`-Enum.
- **LinearGradientBrush:** Definiert ein `Brush`-Objekt mit einem linearen Farbverlauf.
- **PathGradientBrush:** Definiert ein `Brush`-Objekt mit einem Farbverlauf, der durch einen beliebigen Kurvenverlauf (`GraphicsPath`) bestimmt wird. Ein solcher Pinsel entsteht entweder aus einem `GraphicsPath`-Objekt oder einem `Points`- bzw. `PointF`-Array, das die Eckpunkte des Kurvenverlaufs bestimmt.

Grundsätzlich gilt: Wenn Sie eine Linie, einen Linienzug ein Polygonumriss oder einen Kreisumriss zeichnen möchten, verwenden Sie eine der `DrawXXX`-Methoden des `Graphics`-Objektes. Sie benötigen zum Zeichnen in diesen Fall ein `Pen`-Objekt, das die Eigenschaften des verwendeten Stiftes bestimmt.

Möchten Sie eine Fläche füllen, verwenden Sie eine der `FillXXX`-Methoden des `Graphics`-Objektes. Sie benötigen zum Zeichnen dann eines der von `Brush` abgeleiteten Objekte, das die Eigenschaften der Füllung bestimmt.

Angabe von Koordinaten

Die meisten Zeichenfunktionen, die Ihnen das `Graphics`-Objekt zur Verfügung stellt, arbeiten mit der Angabe von Koordinaten, die ihnen entweder als `Single`- oder als `Integer`-Werte übergeben werden. Der Koordinatenursprung liegt in der linken, oberen Ecke des Bildschirms. Neben der Bestimmung von Koordinaten durch einzelne `Single`- oder `Integer`-Werte hält das Framework einige Strukturen bereit, um Sie bei der Angabe von Positionen, Umrissen oder Größen zu unterstützen. Die wichtigsten sind:

- **Point:** Die `Point`-Struktur dient zur Angabe einer Koordinate. Sie übergeben ihr im Konstruktor den X- und Y-Wert als Integerwert.
- **PointF:** Die `PointF`-Struktur dient zur Angabe einer Koordinate mit Fließkommawerten. Sie übergeben ihr im Konstruktor den X- und Y-Wert jeweils als Wert vom Typ `Single`.
- **Size:** Die `Size`-Struktur dient zur Angabe des Ausmaßes eines rechteckigen Objektes. Sie übergeben ihr im Konstruktor die Breite und Höhe als Integer-Wert.
- **SizeF:** Die `SizeF`-Struktur dient zur Angabe des Ausmaßes eines rechteckigen Objektes mit Fließkommawerten. Sie übergeben ihr im Konstruktor die Breite und Höhe jeweils als Wert vom Typ `Single`.
- **Rectangle:** Diese Struktur dient zur Angabe von Position und Ausmaßen eines Rechtecks. Ein `Rectangle` definiert sich durch einen Startpunkt, der als Koordinate angegeben wird, sowie durch die Höhe und die Breite.
- **RectangleF:** Es gilt das zu `Rectangle` Gesagte, nur dass die Werte als Fließkommawert vom Typ `Single` angegeben werden.

Wieso Integer- und Fließkommaangaben für Positionen und Ausmaße?

Das Graphics-Objekt ist nicht auf Pixel als feste Maßeinheit für die Positions- bzw. Größenangaben festgelegt. Vielmehr erlauben die PageUnit-Eigenschaft, die PageScale-Eigenschaft sowie verschiedene Transformationsmethoden eine individuelle Skalierung des Koordinatensystems. Integerwerte sind dann natürlich nicht mehr ausreichend, wenn Sie beispielsweise *Inches* (englische Bezeichnung für die Maßeinheit Zoll, entspricht 2,54 cm) als Maßeinheit bestimmt haben. Auf dem Bildschirm liegen zwischen zwei Pixel, die einen Inch auseinander liegen, natürlich viele weitere Pixel. Die dazwischen liegenden Punkte ließen sich nicht erreichen, könnte man keine gebrochenen Werte für Koordinaten angeben.

Beide Verfahren haben Vor- und Nachteile: Wenn Sie sich für Pixel als Maßeinheit entscheiden (das ist die Standardeinstellung, die durch PageUnit bestimmt wird), können Sie Integer-Werte verwenden; interne Skalierungsumrechnungen entfallen dabei, allerdings können Sie damit nur bestimmte Anwendungen realisieren, wie beispielsweise die Begrenzungen eigener Steuerelemente zeichnen, deren Ausmaße ohnehin in Pixel angegeben werden.

Entscheiden Sie sich für eine andere Maßeinheit, wie beispielsweise Millimeter oder Inch, müssen Sie Fließkommawerte verwenden, um alle Pixel im Koordinatensystem ansteuern zu können. Intern finden natürlich wieder Umrechnungen auf die eigentlichen, physischen Pixelkoordinaten statt, und das kostet einen wenig Rechenzeit. Allerdings ist die Anwendung von Maßeinheiten, die Sie aus der »echten« Welt kennen, für die Umsetzung vieler Anwendungen leichter und flexibler.

Wie viel Platz habe ich zum Zeichnen?

Um die zur Verfügung stehenden Ausmaße eines Formulars (oder einer von Control abgeleiteten Klasse) zu ermitteln, gibt es zwei Möglichkeiten.

- Wenn Ihnen das Graphics-Objekt bekannt ist, verwenden Sie die Nur-Lesen-Eigenschaft VisibleClipbounds des Grafikobjektes. Der Vorteil: Wenn Sie mit der PageUnit-Eigenschaft eine andere Maßeinheit für das Koordinatensystem eingestellt haben, liefert VisibleClipbounds die Ausmaße in den Einheiten zurück, die durch PageUnit eingestellt sind.
- Mit ClientArea des Formulars oder der verwendeten Control-Klasse (oder deren Ableitung) ermitteln Sie den sichtbaren Zeichenbereich in Pixel. Sie benötigen dazu kein Graphics-Objekt.

Das gute, alte Testbild und GDI+ im Einsatz sehen!

Nach so viel grauer Theorie sind Sie sicherlich gespannt, GDI+-Funktionen in der Praxis zu sehen. Seit der Einführung von Kabel- und Satellitenfernsehen Mitte der 80er Jahre können wir uns über eine Unterversorgung mit farbigen Bildern nicht mehr beklagen. Allerdings: Ein bestimmtes Programm ist fast gänzlich von unseren Mattscheiben verschwunden – gemeint ist das gute, alte Testbild. Die Älteren unter Ihnen werden sich sicherlich noch erinnern können, wie es uns erging, wenn wir morgens nicht zur Schule wollten, weil dort entweder eine nicht zu packende Klassenarbeit oder der Kerl aus der 4. Klasse auf uns wartete. Diesem hatten wir tags zuvor sein Pausenbrot »frisiert«, gleichzeitig aber nicht bedacht, dass wir zwar der Held des Tages aber auch ein potentieller Kandidat für die Notaufnahme am nächsten Tag waren. Also hieß es: Fieberthermometer in den Tee, Seifen-

wasser in die Augen (zwei Minuten mit aufgerissenen Lidern ohne zu blinzeln reichten meist auch aus, um die notwendige Augenrötung herbeizuführen) und nach einigen, Oscar-verdächtigen Jammereinlagen ging's ab aufs elterliche Sofa, wo die Flimmerkiste – gerade in Farbe – schon auf ein gut sortiertes Programm wartete: Telekolleg im Dritten, Testbild im Zweiten und Testbild im Ersten.

Und dann hieß es: Sehnsüchtig auf die Sesamstraße um halb zehn warten. Das Testbild, geben Sie es zu, hassten Sie damals sicherlich, wie ich es tat. Und jetzt? Jetzt verbinden wir alle wahrscheinlich kindliche Unbeschwertheit mit exakt dieser Grafik, und wäre es nicht schön, solche Momente nochmals erleben zu können?

Aber: Warum jammern? Wir haben Rechenpower, wir haben Visual Basic und wir haben GDI+! Holen wir es uns zurück!

BEGLEITDATEIEN: Sie finden das folgende Projekt unter `VB 2005 - Entwicklerbuch\G - SmartClient\Kap29\VB-Testbild01`; es trägt den Projektnamen `VBTestbild`, und wenn Sie meiner Generation (so ungefähr 1966–1972) angehören, werden Sie es noch kennen und lieben!

Wenn Sie dieses Programm starten, sehen Sie das Testbild, etwa wie in Abbildung 29.2 gezeigt. Sie können dieses Bild vergrößern und verkleinern, und Sie werden feststellen, dass sich sein Inhalt immer an die aktuelle Größe anpasst. Mal ganz abgesehen davon, dass es heftig flimmert (jedenfalls in dieser Version), ist das der Beweis, dass es sich bei der Grafik nicht um eine simple, im Internet geklaute Bitmap handelt, sondern jeder einzelne Strich, jede einzelne Fläche und jedes Polygon farbig und bunt zur Laufzeit gezeichnet wird. Und glauben Sie mir: Salopp gesagt, war das Kreieren dieses Programms eine tierische Fuckelei, aber ich finde, der Aufwand hat sich gelohnt.

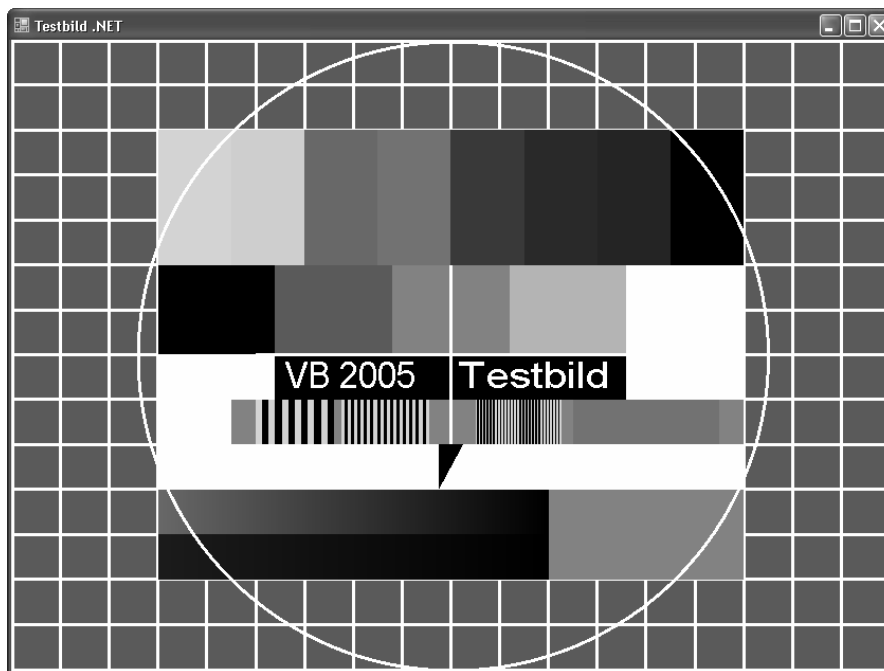


Abbildung 29.2: Für Erinnerungen an die kindliche Unbeschwertheit – das Testbild

Gelohnt, im Übrigen, nicht nur wegen des Ergebnisses, sondern auch, weil das alte Testbild der Öffentlichen-Rechtlichen (ob man damals schon so weit gedacht hat?) viele unterschiedliche Elemente enthielt, die den Einsatz von vielen verschiedenen GDI+-Funktionen erforderlich macht – Sie werden das gleich sehen.

Einige Worte zur Funktionsweise vorweg: Ich habe das Formular so gestaltet, dass es eher als Komponente denn als Formular fungiert. Aus diesem Grund gibt es einige Member-Variablen, zu denen auch Äquivalente als öffentliche Eigenschaften existieren. Das gibt uns zum einen die Möglichkeit, das komplette Formular später abzuleiten und bestimmte Verhaltensweisen durch gezieltes Überschreiben von Eigenschaftenprozeduren zu beeinflussen. Zum anderen können wir aus dem Formular auch ein Steuerelement machen, dessen Verhalten sich durch die öffentlichen Eigenschaften gezielt von außen steuern lässt. Aus Platzgründen finden Sie die meisten dieser Eigenschaften in der vorliegenden Version nicht abgedruckt, da sie die Inhalte der geschützten Member-Variablen, die das Zeichnen eigentlich steuern, nur nach oben durchreichen.

Das Codelisting:

```
Imports System.Drawing.Drawing2D
```

```
Public Class frmMain  
    Inherits System.Windows.Forms.Form
```

Einige Funktionen des Formulars zum Zeichnen werden aus dem Namespace Drawing2D benötigt; deswegen die entsprechende Imports-Anweisung am Anfang des Codes. Die Formalklasse wird aus Form abgeleitet.

```
'Legt fest, ob das Gitter gezeichnet werden soll oder nicht.  
Private myDrawGrid As Boolean  
'Bestimmt, wie das Gitter gezeichnet werden soll.  
Private myGridStyle As GridStyle  
'Bestimmt die Stiftbreite für das Zeichnen des Gitters/Rasters.  
Private myGridLineWidth As Integer  
'Bestimmt die Stiftfarbe für das Zeichnen des Gitters/Rasters.  
Private myGridColor As Color  
'Bestimmt die Hintergrundfarbe.  
Private myBackground As Color  
'Bestimmt die Farben der oberen Balkenreihe.  
Private myBarColors As Color()  
'Bestimmt die Grauschattierungen der darunterliegenden Balkenreihe.  
Private myBarShades As Color()  
'Bestimmt die Grauschattierungen der Balkenreihe, in der sich die Beschriftung befindet.  
Private myBarTitleShades As Color()  
'Definiert die Anzahl der Gitter-/Rasterspalten.  
Private myGridCols As Integer  
'Definiert die Anzahl der Gitter-/Rasterzeilen.  
Private myGridRows As Integer  
'Definiert, in Rasterzeilen gerechnet, den Abstand des "Bildes" von oben.  
Private myUpperOffset As Integer  
'Definiert den Zeichenmodus.  
Private mySmoothingMode As SmoothingMode
```

Die Member-Variablen der Klasse folgen anschließend. Sie bestimmen, in welcher Weise Gitter/Raster und eigentliches Bild gemalt werden sollen. Die Variablen selbst werden im Konstruktor des Programms initialisiert:

```
Public Sub New()  
    MyBase.New()  
  
    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.  
    InitializeComponent()  
  
    ' Initialisierungen nach dem Aufruf InitializeComponent() hinzufügen.  
    DrawGrid = True  
    GridStyle = GridStyle.Lines  
    GridLineWidth = 2  
    GridColor = Color.White  
  
    '18 Spalten und 14 Zeilen  
    GridCols = 18  
    GridRows = 14  
  
    'Bild beginnt in der 3. Reihe.  
    UpperOffset = 2  
  
    'Hintergrund ca. 70% grau  
    Background = Color.FromArgb(90, 90, 90)  
  
    'Farben für die Farbbalken  
    BarColors = New Color() {Color.Pink, Color.GreenYellow, Color.Magenta, Color.Olive, _  
        Color.DarkMagenta, Color.DarkRed, Color.Indigo, Color.Black}  
    'Farben für die darunterliegenden Grauf Flächen  
    BarShades = New Color() {Color.FromArgb(0, 0, 0), _  
        Color.FromArgb(90, 90, 90), _  
        Color.FromArgb(130, 130, 130), _  
        Color.FromArgb(180, 180, 180), _  
        Color.FromArgb(255, 255, 255)}  
    'Farben für die darunterliegenden Grauf Flächen der Titelzeile  
    BarTitleShades = New Color() {Color.White, _  
        Color.Black, _  
        Color.Black, _  
        Color.Black, _  
        Color.White}  
  
    'Fenstertitel  
    Text = "Testbild .NET"  
End Sub
```

Einige Anmerkungen zur Angabe von Farben bei der Verwendung des *Graphics*-Objektes: Die *Color*-Struktur bietet eine elegante und einfache Möglichkeit, Farben zu definieren. Sie verfügt über zahlreiche, statische Funktionen, um Farbwerte anhand von Farbnamen zu ermitteln. Die Farb- und Graubalken, die Sie im Testbild sehen, haben zwar feste Ausmaße, aber Sie können die Anzahl der Balken, die sich in dieser Größenvorgabe befinden, variieren, indem Sie den entsprechenden Arrays Elemente hinzufügen oder aus ihnen entfernen.

```

'Wird aufgerufen, wenn das Bild aus irgendeinem Grund neu gezeichnet werden muss.
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    paintBackground(e.Graphics)
    paintContent(e.Graphics)
End Sub

```

Diese Routine wird automatisch aufgerufen, wenn ein Neuzeichnen des Bildes erforderlich wird. Sie zeichnet zunächst den Hintergrund des Bildes und anschließend das eigentliche Testbild. Diese Routinen sehen folgendermaßen aus:

```

'Hier wird der eigentliche Inhalt gezeichnet.
Private Sub paintContent(ByVal g As Graphics)

    Dim locPen As Pen
    Dim locBrush As Brush
    Dim locBarWidth As Single
    Dim locX As Single
    Dim locRectF As RectangleF

    g.SmoothingMode = mySmoothingMode

    'Raster malen im Bedarfsfall
    If DrawGrid Then
        paintGrid(g)
    End If

    'Figuren malen:

    'obere Balkenreihe mit Farbblöcken...
    locRectF = New RectangleF(GridSize.Width * 3, GridSize.Height * myUpperOffset, _
        GridSize.Width * 12, GridSize.Height * 3)
    locBarWidth = locRectF.Width / BarColors.Length / 1
    For i As Integer = 0 To BarColors.Length - 1
        locBrush = New SolidBrush(myBarColors(i))
        locRectF.Width = locBarWidth
        g.FillRectangle(locBrush, locRectF)
        locRectF.X += locBarWidth
    Next

    'darunter liegende Balkenreihe mit Grauschattierungen...
    locRectF = New RectangleF(GridSize.Width * 3, GridSize.Height * (UpperOffset + 3), _
        GridSize.Width * 12, GridSize.Height * 2)
    locBarWidth = locRectF.Width / BarShades.Length / 1
    For i As Integer = 0 To BarShades.Length - 1
        locBrush = New SolidBrush(myBarShades(i))
        locRectF.Width = locBarWidth
        g.FillRectangle(locBrush, locRectF)
        locRectF.X += locBarWidth
    Next

    'darunter liegende Balkenreihe mit Titel...
    locRectF = New RectangleF(GridSize.Width * 3, GridSize.Height * (UpperOffset + 5), _
        GridSize.Width * 12, GridSize.Height)
    locBarWidth = locRectF.Width / BarShades.Length / 1

```

```

For i As Integer = 0 To BarTitleShades.Length - 1
    locBrush = New SolidBrush(myBarTitleShades(i))
    locRectF.Width = locBarWidth
    g.FillRectangle(locBrush, locRectF)
    locRectF.X += locBarWidth
Next

'darunter komplette zwei Reihen zunächst weiß...
locBrush = New SolidBrush(Color.White)
g.FillRectangle(locBrush, GridSize.Width * 3, GridSize.Height * (UpperOffset + 6), _
    GridSize.Width * 12, GridSize.Height * 2)

'jeweils zwei Farbverlaufsstreifen...
locRectF = New RectangleF(GridSize.Width * 3, GridSize.Height * (UpperOffset + 8), _
    GridSize.Width * 8, GridSize.Height)
locBrush = New LinearGradientBrush(locRectF, Color.Magenta, Color.Black, _
    LinearGradientMode.Horizontal)
g.FillRectangle(locBrush, locRectF)

'der zweite Farbverlaufsstreifen...
locRectF = New RectangleF(GridSize.Width * 3, GridSize.Height * (UpperOffset + 9), _
    GridSize.Width * 8, GridSize.Height)
locBrush = New LinearGradientBrush(locRectF, Color.Blue, Color.Black, LinearGradientMode.Horizontal)
g.FillRectangle(locBrush, locRectF)

'grauer Kasten neben die Farbverläufe...
locBrush = New SolidBrush(Color.FromArgb(130, 130, 130))
g.FillRectangle(locBrush, GridSize.Width * 11, GridSize.Height * (UpperOffset + 8), _
    GridSize.Width * 4, GridSize.Height * 2)

'Geriffele malen...
'Zuerst grauer Kasten als Unterlage
g.FillRectangle(locBrush, GridSize.Width * 4.5F, GridSize.Height * (UpperOffset + 6), _
    GridSize.Width * 10.5F, GridSize.Height)

'dann das linke, größere Geriffele malen...
locRectF = New RectangleF(GridSize.Width * 5, GridSize.Height * (UpperOffset + 6), _
    GridSize.Width * 1.5F, GridSize.Height)

drawAreaCorrugated(g, locRectF, Color.Black, Color.LightGray, 2, GridSize.Width)

'das kleinere Geriffele rechts daneben malen...
locRectF.X = GridSize.Width * 6.75F
locRectF.Width = GridSize.Width * 1.75F
drawAreaCorrugated(g, locRectF, Color.Black, Color.LightGray, 1, GridSize.Width)

'das noch kleinere Geriffele rechts daneben malen...
locRectF.X = GridSize.Width * 9.5F
locRectF.Width = GridSize.Width * 1.75F
drawAreaCorrugated(g, locRectF, Color.Black, Color.LightGray, 0.5F, GridSize.Width)

'das Olivenfarbene daneben...
locRectF.X = GridSize.Width * 11.5F

```

```

locRectF.Width = GridSize.Width * 3
locBrush = New SolidBrush(Color.Olive)
g.FillRectangle(locBrush, locRectF)

'Zielkreuz Kreis in die Mitte...
locPen = New Pen(Color.White, 3)
g.DrawEllipse(locPen, ClientSize.Width \ 2 - ClientSize.Height \ 2, 0, ClientSize.Height, _
    ClientSize.Height)
locPen.Width = 1
g.DrawLine(locPen, GridSize.Width * 9, GridSize.Height * (UpperOffset + 3), _
    GridSize.Width * 9, GridSize.Height * (UpperOffset + 7))
g.DrawLine(locPen, GridSize.Width * 5, GridSize.Height * (UpperOffset + 5), _
    GridSize.Width * 13, GridSize.Height * (UpperOffset + 5))

'das kleine Dreieck in den weißen Zwischenraum...
Dim locPoints(2) As PointF
locPoints(0) = New PointF(GridSize.Width * 8.75F, GridSize.Height * (UpperOffset + 7))
locPoints(1) = New PointF(locPoints(0).X, GridSize.Height * (UpperOffset + 8))
locPoints(2) = New PointF(GridSize.Width * 9.25F, locPoints(0).Y)
g.FillPolygon(New SolidBrush(Color.Black), locPoints)

'Texte hineinschreiben.
drawStringInFrame(g, New SolidBrush(Color.White), "VB.NET", "Arial", _
    New RectangleF(GridSize.Width * 5.5F, GridSize.Height * (UpperOffset + 5), _
    GridSize.Width * 3, GridSize.Height))

drawStringInFrame(g, New SolidBrush(Color.White), "Testbild", "Arial", _
    New RectangleF(GridSize.Width * 9, GridSize.Height * (UpperOffset + 5), _
    GridSize.Width * 3.5F, GridSize.Height))

End Sub

```

Diese Prozedur bildet den Kern des Programms – sie sorgt dafür, dass das Bild auch tatsächlich auf dem Bildschirm erscheint. Anhand dieser Routine können Sie sehen, wie Füllungen und Linienfiguren mit dem GDI+ gezeichnet werden, und Sie können ebenfalls erkennen, wie einfach die Handhabung von Grafikfunktionen mit dem GDI+ im Grunde genommen ist.

Komplexe Grafikfunktionen, wie beispielsweise das Zeichnen des »geriffelten« Bereiches oder das Platzieren des Textes in der richtigen Größe, sind in verschiedene Unterrouninen ausgelagert, die Sie im Folgenden beschrieben finden:

```

'Geriffelten Bereich malen.
Private Sub drawAreaCorrugated(ByVal g As Graphics, ByVal rectF As RectangleF, _
    ByVal col1 As Color, ByVal col2 As Color, _
    ByVal relCellStepWidth As Single, ByVal relCellWidth As Single)

    Dim locPCol1 As New SolidBrush(col1)
    Dim locPCol2 As New SolidBrush(col2)
    Dim locCurrentBrush As Brush
    Dim locAltFlag As Boolean = False
    Dim locStep As Single = relCellWidth / (1 / relCellStepWidth * 15)
    For x As Single = rectF.X To rectF.Right Step locStep
        locCurrentBrush = DirectCast(If(locAltFlag, locPCol1, locPCol2), SolidBrush)
        g.FillRectangle(locCurrentBrush, x, rectF.Y, locStep, rectF.Height)
    Next

```

```

        locAltFlag = Not locAltFlag
    Next

End Sub

'Zeichnet den Text so, dass er genau in ein Rechteck passt.
Private Sub drawStringInFrame(ByVal g As Graphics, ByVal brush As Brush, ByVal text As String, _
    ByVal fontName As String, ByVal rectF As RectangleF)

    'Skalierungseinstellungen zum Wiederherstellen speichern.
    Dim locGState As GraphicsState = g.Save
    'Font mit 12 Pt. Höhe aus Fontnamen anlegen.
    Dim locFont As New Font(fontName, 12, FontStyle.Regular)
    'Ausmaße des Strings messen.
    Dim locSize As SizeF = g.MeasureString(text, locFont)
    'Faktoren für die Skalierung ermitteln, sodass der String...
    Dim locScaleV As Single = rectF.Height / locSize.Height
    '...genau in das angegebene Rechteck passt.
    Dim locScaleH As Single = rectF.Width / locSize.Width
    'Koordinatensystem verschieben
    g.TranslateTransform(rectF.X, rectF.Y)
    'Koordinatensystem neu skalieren
    g.ScaleTransform(locScaleH, locScaleV)
    'String im skalierten Koordinatensystem ausgeben.
    g.DrawString(text, locFont, brush, 0, 0)
    'Alte Skalierungs- und Transformationseinstellungen wiederherstellen.
    g.Restore(locGState)

End Sub

```

Diese letzte Prozedur demonstriert den Einsatz von Text- und Skalierungsfunktionen. Für deren genaues Verständnis ist allerdings ein klein wenig mehr Hintergrundwissen erforderlich, das Ihnen der folgende kurze Exkurs liefert.

Exaktes Einpassen von Text mit GDI+-Skalierungsfunktionen

Um eine Zeichenkette in ein Graphics-Objekt auszugeben, benötigen Sie neben einem Font-Objekt, das den zu verwendenden Zeichensatz bestimmt, auch ein Brush-Objekt, das die Pinseleigenschaften darstellt und damit Farbe und Füllung definiert, mit denen der Text gezeichnet wird. Für die Größenbestimmung der auszugebenden Zeichenkette ist normalerweise ausschließlich die Größe des Zeichensatzes verantwortlich; GDI+ bietet Ihnen standardmäßig keine Möglichkeit, einen auszugebenden Text automatisch auf eine bestimmte Höhe oder Breite zu skalieren. Wenn Sie einen Text mit der DrawString-Funktion in ein Graphics-Objekt hineinschreiben, hat er damit genau die Größe, die sich durch das angegebene Font-Objekt und die Breite der im auszugebenden Text enthaltenen Buchstaben ergibt.

Allerdings bietet das GDI+ eine Reihe von Skalierungsfunktionen, mit denen das Koordinatensystem in beide Richtungen gedehnt oder gestaucht werden kann. ▶

In Zusammenarbeit mit der `MeasureString`-Funktion, die die Ausmaße eines Textes ermitteln kann, *ohne* ihn dabei wirklich zu auszugeben, erlaubt das die genaue Dehnung/Stauchung des Koordinatensystems, sodass der Text – egal mit welchen Fonteeinstellungen gezeichnet – exakt in die Ausmaße eines bestimmaren rechteckigen Bereichs eingepasst werden kann. Die Verfahrensweise dabei ist relativ einfach:

Der Text wird zunächst mit `MeasureString` vermessen; dabei werden seine Höhe und seine Breite in Pixel³ ermittelt. Die Skalierungsfaktoren für den Text ergeben sich jetzt aus einer simplen Division der Ausmaße des Zielrechteckes durch die Ausmaße des den Text tatsächlich umschließenden Rechtecks.

Damit die Skalierung für das `Graphics`-Objekt später wieder auf seine Ursprungseinstellung zurückgestellt werden kann, kann die aktuelle Skalierungseinstellung mit der `Save`-Methode in einem so genannten `GraphicsState`-Objekt gespeichert werden. Erst jetzt werden die errechneten Skalierungsfaktoren mithilfe der Methode `TranslateTransform` für das aktuelle `Graphics`-Objekt bestimmt. Da der zuvor gemessene Text derselbe ist, der nun mit `DrawString` in das `Graphics`-Objekt ausgegeben wird, und die Skalierung durch Anwenden von `TranslateTransform` umgestellt wurde, passt er exakt in das Zielrechteck.

Damit alle weiteren Zeichenfunktionen unskaliert stattfinden können, wird die ursprüngliche Skalierung durch die `Restore`-Methode zu guter Letzt wieder in den Ausgangszustand zurückversetzt.

Die einzige Zeichenroutine, die jetzt noch fehlt, ist die zum Zeichnen des Gitters. Im folgende Code-Listing werden Sie feststellen: Die `paintGrid`-Methode ist so ausgelegt, dass sie wahlweise ein Raster oder ein Gitter zeichnen kann. Ein Rasterpunkt besteht dabei allerdings nicht aus einem einzelnen Pixel, sondern aus einem kleinen Rechteck. Das GDI+ stellt keine Funktion zur Verfügung, mit der ein einzelner Punkt gezeichnet werden kann – das ist der Hintergrund. Alternativ könnten Sie auch eine Linie mit gleichem Anfangs- und Endpunkt zeichnen, das Ergebnis wäre ähnlich:

```
'Zeichnet das Raster oder das Gitter.
Private Sub paintGrid(ByVal g As Graphics)
    Dim locPen As New Pen(GridColor, GridLineWidth)
    Dim locBrush As New SolidBrush(GridColor)

    If GridStyle = GridStyle.Dots Then
        For x As Single = 0 To ClientSize.Width Step GridSize.Width
            For y As Single = 0 To ClientSize.Height Step GridSize.Height
                g.FillRectangle(locBrush, x, y, GridLineWidth, GridLineWidth)
            Next
        Next

    Else
        For x As Single = 0 To ClientSize.Width Step GridSize.Width
            g.DrawLine(locPen, x, 0, x, ClientSize.Height)
        Next
    End If
End Sub
```

³ Vorausgesetzt, Sie haben die verwendete Maßeinheit für das aktuelle `Graphics`-Objekt nicht zuvor mit seiner `PageUnit`-Eigenschaft auf einen anderen Wert gesetzt.

```

        For y As Single = 0 To ClientSize.Height Step GridSize.Height
            g.DrawLine(1ocPen, 0, y, ClientSize.Width, y)
        Next
    End If
End Sub

```

Übrigens: Wenn Sie das Formular vergrößern oder verkleinern, löst das nicht notwendigerweise ein Paint-Ereignis aus. Das Paint-Ereignis wird nur beim Vergrößern des Formulars ausgelöst, und Sie können mit dem Graphics-Objekt, das jetzt übergeben wird, nur den Bereich erneuern, der durch das Vergrößern auch wirklich neu gezeichnet werden müsste. Das ergibt Sinn bei Inhalten, die statisch sind – also nicht durch die Größe des Formulars beeinflusst werden. Bei einer Tabellenkalkulation beispielsweise ist der dargestellte Inhalte eines Fensters nicht von seiner Größe abhängig; nur der dargestellte Ausschnitt verändert sich mit dem Vergrößern des Fensters.

In unserem Beispiel ist das anders. Wenn das Formular vergrößert oder verkleinert wird, muss der *komplette* Formularinhalt neu gezeichnet werden. Aus diesem Grund schaut die Prozedur, die dieses Ereignis verarbeitet, folgendermaßen aus:

```

'Wird aufgerufen, wenn das Formular in seiner Größe verändert wird.
Protected Overrides Sub OnResize(ByVal e As System.EventArgs)
    Me.Invalidate()
End Sub

```

Invalidate bewirkt, dass der gesamte Bereich auf den es angewendet wird, hier Me – also das Formular selbst –, als »ungültig« gekennzeichnet wird. Dabei wird dann das Paint-Ereignis ausgelöst. Der Formularinhalt wird also komplett neu gezeichnet.

Und jetzt, nachdem Sie wissen, wie das Programm funktioniert: Stürzen Sie sich hinein in den Quellcode, und experimentieren Sie mit den Einstellungen, die im Konstruktor des Formulars vorgenommen werden. Sie werden sehen, dass Sie durch Ausprobieren der verschiedenen Einstellungen der Objekte, die Graphics anbietet, am besten den Umgang mit dem GDI+ lernen können!

Flimmerfreie, fehlerfreie und schnelle Darstellungen von GDI+-Zeichnungen

Das Beispielprogramm aus dem vorherigen Abschnitt hat die Leistungsfähigkeit des GDI+ eindrucksvoll demonstriert. Allerdings hat es ebenso gezeigt, dass noch einige Handgriffe notwendig sind, um die Darstellung wirklich zu perfektionieren, denn:

- Die Bilddarstellung flimmert heftig, wenn der Anwender das Testbild vergrößert oder verkleinert.
- Das Formular sollte sich nur proportional vergrößern lassen (Festes Seitenverhältnis in X- und Y-Richtung), damit man feststellen kann, ob der Monitor einen Kreis auch wirklich rund darstellt. Das Seitenverhältnis sollte sich durch eine Eigenschaft einstellen lassen.
- Der Bildaufbau ist gerade beim Vergrößern und Verkleinern des Formulars vergleichsweise langsam.
- Und falls Sie ganz genau hingeschaut haben, werden Sie bemerkt haben, dass die Linienstärke bei den äußeren Linien und bei den Berührungspunkten des Kreises mit den Formularrändern berücksichtigt werden sollte.

Diese Liste von Unzulänglichkeiten ist typisch für grafische Inhalte, die in Fenstern unter Windows dargestellt werden. Selbst ein Programm wie der Windows-Explorer, von dem man meinen könnte, es sei nach Jahren wirklich ausgereift, flimmert beim Vergrößern oder Verkleinern munter vor sich hin.⁴ Die nächsten Abschnitte sollen Ihnen helfen, die richtige Vorgehensweise zu finden, um Ihren Formularen und (später auch) Steuerelementen ein professionelles Aussehen zu verleihen.

Zeichnen ohne Flimmern

Bei dem ersten Problem hilft das Framework mit einem Prinzip, nach dem man schon seit Jahren und nicht erst seit Windows verfährt, um Flimmern bei der Darstellung von bewegten Bildern zu vermeiden. Die Technik ist so simpel wie genial: Anstatt ein Bild zu löschen und es verändert neu zu zeichnen, komplettiert man es zunächst in einer unsichtbaren Bitmap im Speicher. Wenn das Bild komplett gezeichnet wurde, kopiert man es als Ganzes in den sichtbaren Anzeigebereich. So ist das eigentliche Entstehen des Bildes unsichtbar und damit flimmerfrei. Darüber hinaus muss das Bild für das Neuzeichnen nicht gelöscht werden (was ein Großteil des Flimmerns verursacht), denn wenn eine entsprechende Instanz das komplett fertige Bild in das zu überschreibende, sichtbare Bild kopiert, wird sowieso jeder einzelne Pixel des alten Bildes gelöscht. Ein Löschen des Hintergrunds wäre also total überflüssig.

Wie schon angedeutet, müssen Sie diese Funktionalität nicht selber implementieren, da sie das Framework schon fix und fertig bereithält. Sie müssen dem Framework lediglich mitteilen, dass Sie die Sonderbehandlung des »Hintergrund-Löschen-Ereignisses«, das von Windows ausgelöst wird, nicht wünschen, sondern die gerade beschriebene Methode – sie nennt sich auf neudeutsch *Double Buffering* (Doppelpufferung) – anwenden möchten.

BEGLEITDATEIEN: Sie finden das modifizierte Testbild-Projekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - Smart-Client\Kap29\VBTestbild02\`

Sie erreichen das mit zwei simplen Zeilen, die Sie im Konstruktor hinterlegen:

```
Public Sub New()  
    MyBase.New()  
  
    ' Dieser Aufruf ist für den Windows Form-Designer erforderlich.  
    InitializeComponent()  
  
    Me.SetStyle(ControlStyles.DoubleBuffer, True)  
    Me.SetStyle(ControlStyles.AllPaintingInWmPaint, True)
```

Wenn Sie das Programm nach dieser Modifizierung starten, ist es vorbei mit dem Flimmern. Sie können das Formular nach Belieben vergrößern oder verkleinern, ohne beim Neuaufbau zuschauen zu müssen.

⁴ Achten Sie mal beim Vergrößern oder Verkleinern auf das *TreeView*-Steuerelement des Explorers, das die Laufwerke darstellt.

HINWEIS: Neu in Visual Basic 2005 bzw. im Framework 2.0 ist eine Eigenschaft der Basisklasse `Control` namens `DoubleBuffered`. Das Setzen dieser Eigenschaft bewirkt letzten Endes dasselbe wie `Me.SetStyle(ControlStyles.DoubleBuffer, True)` zusammen mit `Me.SetStyle(ControlStyles.AllPaintingInWmPaint, True)`. Der neu eingeführte Stil `ControlStyles.OptimizedDoubleBuffer` verspricht mehr, als er hält: Das Durchführen des internen Double Buffering funktioniert auf die gleiche Weise, wie beim Einstellen des Double Buffering auf die ursprünglichen Art über die beiden `SetStyle`-Methoden. Es ist also anzunehmen, dass das `OptimizedBuffering`-Flag nur aus Abwärtskompatibilitätsgründen eingeführt wurde, denn nur auf dieses Flag wird Framework-intern durch die `DoubleBuffered`-Eigenschaft zurückgegriffen.

Eigenschaften von Formularen und Control-Ableitungen mit `SetStyle` definieren

`SetStyle` dient übrigens nicht nur dazu, das *Double Buffering* ein- und auszuschalten. Auch andere Verhaltensweisen einer von `Control` abgeleiteten Klasse (dazu gehört auch ein Formular) lassen sich mit dieser Methode steuern.

Die grundsätzliche Verwendung von `SetStyle` funktioniert folgendermaßen:

```
Me.SetStyle(ControlStyles.Style1 or ControlStyle2 or ..., True|False)
```

Als ersten Parameter übergeben Sie `SetStyle` eine Kombination aus Elementen der `ControlStyles`-Enum. Der zweite Parameter bestimmt, ob die entsprechenden Flags ein- bzw. ausgeschaltet werden sollen. Welche Flags dabei welche Aufgaben übernehmen, zeigt die folgende Tabelle. Bitte verwenden Sie in diesem Falle nicht die Visual Studio-Online-Hilfe, da sie die Aufgaben der einzelnen Flags recht schwammig, teilweise sogar missverständlich erklärt.

Member-Name	Wert	Beschreibung
<code>ContainerControl</code>	1	Die auf <code>Control</code> basierende Komponente ist ein <i>Container</i> -Control. Dieses Flag ist automatisch gesetzt, wenn Sie eine Klasse aus <i>ScrollableControl</i> ableiten.
<code>UserPaint</code>	2	Definiert, dass die Ereignisse <i>OnPaint</i> und <i>OnBackgroundPaint</i> von <i>WndProc</i> ausgelöst werden. Dieses Flag ist standardmäßig gesetzt. Wenn Sie dieses Flag löschen, müssen Sie <i>WndProc</i> überschreiben und die entsprechenden Nachrichten auswerten, um die notwendigen Maßnahmen zum Zeichnen des Fensterinhaltes zu ergreifen.
<code>Opaque</code>	4	Wenn Sie dieses Flag setzen, wird <i>OnPaintBackground</i> nicht ausgelöst und der Hintergrund damit nicht gezeichnet.
<code>ResizeRedraw</code>	16	Bestimmt, dass <i>OnResize</i> , das bei Größenveränderung des Controls/Formulars automatisch aufgerufen wird, ein <i>Invalidate</i> auslöst und damit automatisch das Neuzeichnen des Client-Bereichs erzwingt.
<code>FixedWidth</code>	32	Formulare lassen sich durch die <i>AutoScale</i> -Eigenschaft so einrichten, dass sie sich automatisch an eine neue Schriftart anpassen, die ihnen durch die <i>Font</i> -Eigenschaft zugewiesen wird. Dadurch wird das Formular selbst und auch alle in ihm enthaltenen Steuerelemente entsprechend der Größe des verwendeten Fonts vergrößert oder verkleinert. ⁵ Möchten Sie verhindern, dass die Skalierung der Formularbreite stattfindet, setzen Sie dieses Flag. Mit diesem Flag können Sie <i>nicht</i> festlegen, dass ein Formular oder Control in seiner Breite nicht verändert werden darf. ►

⁵ Beachten Sie, dass dieses Verhalten nur beim Erstellen des Formulars funktioniert, also wenn der Font noch im Konstruktor verändert wird.

Member-Name	Wert	Beschreibung
FixedHeight	64	Es gilt das für <i>FixedWidth</i> Gesagte, nur für die Höhe des Formulars.
StandardClick	256	Dieses Flag ist standardmäßig gesetzt. Wenn Sie möchten, dass das Control oder das Formular kein <i>Click</i> -Ereignis auslösen soll, löschen Sie dieses Flag. WICHTIG: Wenn Sie dieses Flag löschen, löst das Control/Formular auch kein <i>DoubleClick</i> -Ereignis mehr aus.
Selectable	512	Dieses Flag ist für Formulare und Controls standardmäßig gesetzt. Es bestimmt, ob entsprechende Ereignisse von Windows überhaupt zur Fokussierung des Controls/Formulars führen können. Beachten Sie, dass <i>Container Controls (ScrollableControl, ContainerControl)</i> den Fokus nicht erhalten können, wenn sie andere Controls beinhalten. Das gilt unabhängig von der Einstellung dieses Flags. Beachten Sie auch, dass von <i>Control</i> abgeleitete Klasseninstanzen standardmäßig die <i>Fokusbenachrichtigung durch Mausclickaktivierung nicht erhalten</i> , wenn das folgende <i>UserMouse</i> -Flag nicht gesetzt ist.
UserMouse	1024	Dieses Flag muss gesetzt werden, damit eine von Control abgeleitete Instanz Fokussierungsnachrichten erhalten kann, wenn diese durch Mausclick ausgelöst wurden. Bitte beachten Sie, dass entgegen den Angaben in der Online-Hilfe das Setzen dieses Flags nicht dazu führt, dass Sie Mausereignisse von Grund auf neu implementieren müssen! <i>OnMouseXXX</i> , <i>OnClick</i> und <i>OnDoubleClick</i> werden nach wie vor ausgeführt!
SupportsTransparentBackColor	2048	Direkte Ableitungen von <i>Control</i> unterstützen normalerweise keine transparenten Hintergrundfarben. Mit Hilfe dieses Flags können Sie die Unterstützung explizit einschalten. Anschließend akzeptiert die <i>Control</i> -Ableitung zum Simulieren von Transparenz eine Hintergrundfarbe mit einer Alpha ⁶ -Komponente, die kleiner als 255 ist. Für Formulare ist dieses Flag standardmäßig gesetzt; Sie können die Farbe, die die Transparenz bestimmt, mit der <i>TransparencyKey</i> -Eigenschaft definieren.
StandardDoubleClick	4096	Dieses Flag ist standardmäßig gesetzt. Wenn Sie möchten, dass das Control oder das Formular kein <i>DoubleClick</i> -Ereignis auslösen sollen, löschen Sie dieses Flag. Auch wenn Sie das <i>StandardClick</i> -Flag löschen, erhält das Control/Formular kein <i>DoubleClick</i> -Ereignis mehr.
AllPaintingInWmPaint	8192	Normalerweise löst Windows für Controls und Formulare eine <i>WM_ERASEBKGD</i> -Nachricht bei der Notwendigkeit des Neuzeichnens eines Client-Bereichs aus. Die Behandlung dieser Nachricht führt dazu, dass der Client-Bereich gelöscht – also mit einer bestimmten Farbe komplett ausgefüllt wird. In vielen Fällen führt das zu unerwünschten Flimmereffekten beim Neuzeichnen des Client-Bereichs. Setzen Sie dieses Flag, ignoriert das Steuerelement die <i>WM_ERASEBKGD</i> -Fenstermeldung, um das Flimmern zu verringern.
CacheText	16384	Setzen Sie dieses Flag, bewahrt das Steuerelement eine Kopie des Textes auf, sodass dieser nicht jedes Mal, wenn er benötigt wird, aus dem Windows-Fenster abgerufen werden muss. Dieses Flag ist standardmäßig nicht gesetzt. Dieses Verhalten verbessert die Leistung, erschwert jedoch die Textsynchronisierung.
EnableNotifyMessage	32768	Setzen Sie dieses Flag, wird <i>OnNotifyMessage</i> für jede Meldung aufgerufen, die aus der Nachrichtenwarteschlange an <i>WndProc</i> des Controls bzw. Formulars gesendet wird. Dieses Flag ist standardmäßig nicht gesetzt. ►

⁶ Die Alpha-Komponente bei einer Farbangabe bestimmt die »Durchscheinstärke«. 255 entspricht »voll deckend«, 0 entspricht »voll durchscheinend«.

Member-Name	Wert	Beschreibung
DoubleBuffer	65536	Schaltet das DoubleBuffering ein. Parallel dazu sollten Sie <i>AllPaintingInWmPaint</i> ebenfalls setzen, damit das <i>Double Buffering</i> in Kraft treten kann.
OptimizedDoubleBuffer	131072	Wenn Sie diese Eigenschaft auf True festlegen, müssen Sie auch <i>AllPaintingInWmPaint</i> auf True festlegen, um das Double Buffering beim Zeichnen zu aktivieren. Hinweis: Dieses Flag wird von der im Framework 2.0 neu vorhandenen <i>DoubleBuffered</i> -Eigenschaft verwendet und ist vermutlich nur aus Gründen der Abwärtskompatibilität vorhanden, da beim Zeichnen eines Steuerelements in seiner internen <i>WmPaint</i> -Routine bei gesetztem <i>OptimizedDoubleBuffer</i> -Flag nichts anderes passiert, als beim Setzen des »einfachen« <i>DoubleBuffer</i> -Flags.
UseTextForAccessibility	262144	Gibt an, dass der Wert der Text-Eigenschaft bei Steuerelementen, sofern festgelegt, den Namen und die Tastenkombination für aktive Eingabehilfen des Steuerelements bestimmt.

Tabelle 29.1: Die Einstellungen der ControlStyles-Enum

Mit dem Wissen um die Funktionsweisen dieser Flags können wir mit einem kleinen Handgriff das Programm weiter optimieren. Durch das Setzen des Flags *ResizeRedraw* kümmert sich schon die Basisklasse um den Aufruf von *Invalidate* beim Vergrößern oder Verkleinern des Formulars. Durch Einfügen einer weiteren Zeile

```
Me.SetStyle(ControlStyles.ResizeRedraw, True)
```

im Konstruktor des Programms wird damit die Behandlung des kompletten *Resize*-Ereignisses überflüssig. Es ist in dieser Version des Programms auch nicht mehr vorhanden.

Was das Thema Geschwindigkeit anbelangt: Wenn Sie das Programm in dieser Version starten, bemerken Sie, dass es gar nicht so langsam läuft, wie es ursprünglich den Anschein hatte. Durch das Eliminieren des Flimmerns erkennen Sie die wahre Geschwindigkeit, mit der man auch auf langsameren Maschinen durchaus leben kann.

Programmtechnisches Bestimmen der Formulargröße

Es gibt zahlreiche Anwendungen, bei denen es notwendig ist, ein Formular oder ein Steuerelement zur Laufzeit auf eine bestimmte Breite oder eine bestimmte Höhe zu beschränken. In unserem Beispiel ergibt es beispielsweise Sinn, eine Eigenschaft einzuführen, die das Seitenverhältnis reglementiert. Nur wenn das Seitenverhältnis des Formulars auch dem Seitenverhältnis des verwendeten Monitors entspricht, können Sie beurteilen, ob der Monitor einen Kreis auch wirklich als Kreis darstellt.

Mit der *SetBoundsCore*-Methode eines Formulars oder eines *Control*-Derivats bestimmen Sie dessen Ausmaße. Allerdings können Sie die Größeneinstellungen nicht im *Resize*-Ereignis vornehmen, da das Neupositionieren des Fensters schon vor dem Auslösen des Ereignisses geschieht. Das Ergebnis wäre ein heftiges Flimmern. Was wir bräuchten, wären weitere Ereignisse, mit denen wir erkennen könnten, wann ein bestimmter Vorgang wie das Verschieben oder das Vergrößern bzw. Verkleinern eines Fensters beginnt und wann er abgeschlossen ist.

Seit der Version 2.0 des Frameworks gibt es zwei Ereignisse, die Sie dabei unterstützen. Diese Ereignisse nennen sich *ResizeBegin* und *ResizeEnd*. Wenn das Vergrößern eines Formulars abgeschlossen

wurde, wird das `ResizeEnd`-Ereignis ausgelöst, und wir können das Formular hier auf das richtige Seitenverhältnis einstellen.

Leider verhält sich diese Sache nicht ganz so einfach. Denn das `ResizeEnd`-Ereignis wird auch dann ausgelöst, wenn Sie das Formular nur verschoben haben. Warum das so ist, ist mir ehrlich gesagt schleierhaft – im *Product Feedback Center* von Visual Studio gibt es aber bereits einen Vorschlag, das Verhalten dieses Ereignisses zu überarbeiten.

Um dem Programm abzugewöhnen, das Formular auch beim Verschieben anzupassen, behelfen wir und deswegen mit einem kleinen Trick, der deutlich wird, wenn Sie sich die Implementierung der entsprechenden Codezeilen anschauen.

BEGLEITDATEIEN: Sie finden das modifizierte Testbild-Projekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - Smart-Client\Kap29\VBTestbild03\`

```
'Wird zu Beginn einer Größenänderung aufgerufen
Protected Overrides Sub OnResizeBegin(ByVal e As System.EventArgs)
    MyBase.OnResizeBegin(e)
    'Das Flag zurücksetzen, das festhält, dass das Formular verschoben wurde
    myJustMoved = False
End Sub

'Wird aufgerufen, wenn eine Größenänderung abgeschlossen wurde
Protected Overrides Sub OnResizeEnd(ByVal e As System.EventArgs)
    'Nach Abschluss des Resize-Vorgangs Seitenverhältnis berücksichtigen,
    'aber nur, wenn dieses Ereignis nicht durch das Verschieben des Formulars
    'ausgelöst wurde.
    If Not myJustMoved Then
        'Neue Größe des Formulars im richtigen Seitenverhältnis:
        MyBase.SetBoundsCore(Location.X, Location.Y, _
            Cint(Size.Height * XYRatio), Size.Height, BoundsSpecified.Width)
    End If
    MyBase.OnResizeEnd(e)
End Sub

'Wird beim Verschieben des Formulars aufgerufen
Protected Overrides Sub OnMove(ByVal e As System.EventArgs)
    MyBase.OnMove(e)
    'Das Formular wurde verschoben; ResizeEnd wird am Ende auch
    'ausgelöst, darf aber nicht berücksichtigt werden!
    myJustMoved = True
End Sub
```

Mit dieser Änderung können wir die Möglichkeit zur Einstellung des Seitenverhältnisses nun als Eigenschaft in die neue Programmversion einbauen und im nunmehr vorhandenen `ResizeEnd`-Ereignis dafür sorgen, dass das Formular nach Abschluss des Größenveränderungsvorgangs richtig positioniert wird:

```

Public Property XYRatio() As Single
    Get
        Return myXYRatio
    End Get
    Set(ByVal Value As Single)
        myXYRatio = Value
    End Set
End Property

```

Diese Eigenschaft wird im Konstruktor auf ein Seitenverhältnis von 4 : 3 gesetzt – wie es den meisten Monitoren entspricht:⁷

```

Public Sub New()
    .
    .
    .
    'Seitenverhältnis definieren
    XYRatio = 4 / 3

    'für 19"-TFTs mit 1280x1024:
    'XYRatio = 5 / 4
    .
    .
    .

```

Was Sie beim Zeichnen von breiten Linienzügen beachten sollten

Wenn Sie das Testbild in Abbildung 29.2 genauer betrachten, werden Sie feststellen, dass die äußeren Linien nicht richtig zu sehen sind. Das gilt für die Linien des Rasters genau so wie für die Berührungspunkte des Kreises. Die Gründe dafür soll das folgende kleine Projekt demonstrieren.

BEGLEITDATEIEN: Sie finden dieses Projekt für dieses Beispiel im Verzeichnis `.\\VB 2005 - Entwicklerbuch\\G - Smart-Client\\Kap29\\PenWidthDemo01\\`

Wenn Sie das Programm starten, sehen Sie im Formular eine Grafik, die in etwa der Abbildung 29.3 entspricht.

⁷ Ausnahmen bilden 17“, 19“ und 20“-TFT-Displays mit einer Auflösung von 1280 x 1024 Pixel, die ein Seitenverhältnis von 5 : 4 aufweisen.

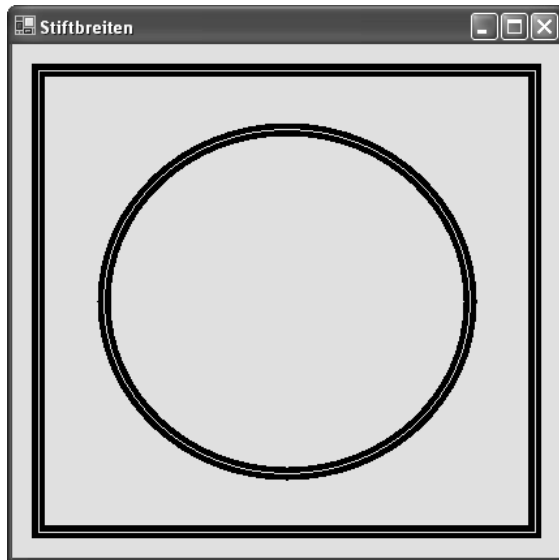


Abbildung 29.3: Wenn Sie Stiftstärken größer als ein Pixel verwenden, dann müssen Sie davon ausgehen, dass die weiteren Pixel einer Linie gleichmäßig um den eigentlich Pixel herum verteilt sind. In dieser Grafik haben die gelben und schwarzen Figuren die gleichen Ausmaße, aber andere Stiftstärken.

Die Abbildung zeigt es deutlich: Die Pixel des breiteren Stiftes der schwarzen Figuren verteilen sich um die ein Pixel breiten Linien des gelben Stiftes, und zwar auf allen Seiten zu genau gleichen Teilen. Wenn Sie also beispielsweise einen Rahmen voll sichtbar in den Client-Bereich eines Formulars zeichnen wollen, dann müssen Sie im Falle des Rechtecks die Hälfte der Linienstärke in die Koordinaten mit einrechnen. Zur Verdeutlichung: Der Code, der diese Figuren ins Formular malt, sieht folgendermaßen aus:

```
'Zeichnet jeweils ein Rechteck in einer dicken, schwarzen und einer dünnen, gelben Umrandung.
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
    Dim locSchwarzerStift As New Pen(Color.Black, 10)
    Dim locGelberStift As New Pen(Color.Yellow, 1)
    Dim locFürRechteck As New Rectangle(20, 20, _
        ClientSize.Width - 40, ClientSize.Height - 40)
    Dim locOffsetDrittel As New Size(ClientSize.Width \ 6, ClientSize.Height \ 6)
    Dim locFürKreis As New Rectangle(locOffsetDrittel.Width, _
        locOffsetDrittel.Height, _
        ClientSize.Width - 2 * locOffsetDrittel.Width, _
        ClientSize.Height - 2 * locOffsetDrittel.Height)
    e.Graphics.DrawRectangle(locSchwarzerStift, _
        locFürRechteck)
    e.Graphics.DrawRectangle(locGelberStift, _
        locFürRechteck)
    e.Graphics.DrawEllipse(locSchwarzerStift, _
        locFürKreis)
    e.Graphics.DrawEllipse(locGelberStift, _
        locFürKreis)
End Sub
```

Noch problematischer wird es, wenn Sie Linienzüge aufbauen wollen – beispielsweise, wenn Sie ein zu einer Seite offenes Dreieck aus drei verschiedenen Linien zusammensetzen müssen; beim Zeichnen von breiten Linienzügen gibt es nämlich unschöne Überschneidungen, wenn diese aus einzelnen Linien aufgebaut sind, wie das folgende Beispiel zeigt:

BEGLEITDATEIEN: Sie finden dieses Projekt im Verzeichnis `.\VB 2005 - Entwicklerbuch\G - Smart-Client\Kap29\PenWidthDemo02\`

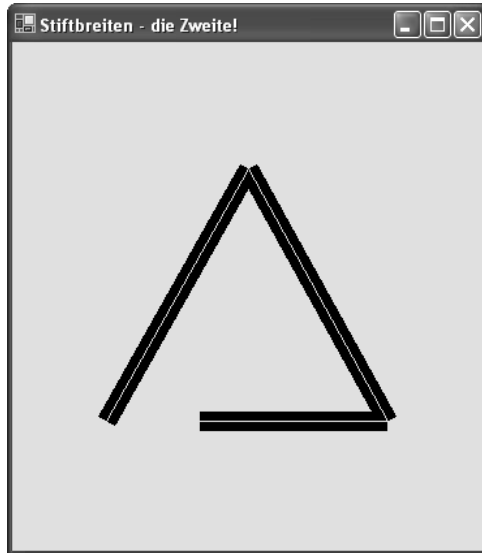


Abbildung 29.4: Bei unabhängigen Linien, die Linienzüge bilden sollen, ist das Verhalten bei großen Stiftstärken umso störender

In diesem Beispiel sollen drei Linien zu einer Figur – in diesem Falle zu einem zu einer Seite offenen Dreieck – verbunden werden; leider klappt das nur ansatzweise: Während es bei den inneren, ein Pixel breiten Linien keine Probleme gibt, sind die äußeren Linien als nicht wirklich miteinander verbunden erkennbar. Der Grund: Die jeweils nächste Linie weiß nichts davon, dass eine weitere Linie folgt, und dass die offenen Zwischenräume mit der Stiftfarbe (oder dem Muster, falls der Stift aus einem Pinsel entstanden ist) aufgefüllt werden sollten.

Geschlossene Figuren mit Polygon und GraphicsPath

Um dieses Manko zu beheben, bietet das GDI+ das so genannte `GraphicsPath`-Objekt an. Das `GraphicsPath`-Objekt erlaubt das Erstellen von Linienzügen, die wirklich miteinander verbunden sind, und seine Verwendung ist denkbar einfach. Zu jeder herkömmlichen `DrawLine`-Methode des GDI+ gibt es ein Äquivalent, mit dem Sie einem `GraphicsPath` einen Linienzug hinzufügen können.

Angenommen, Sie haben mit der Anweisung

```
Dim lcnLinienverbund As New GraphicsPath
```

ein neues `GraphicsPath`-Objekt erstellt. In diesem Fall verwenden Sie zum Zeichnen einer Linie nicht die `DrawLine`-Methode, die Sie direkt auf das `Graphics`-Objekt anwenden, sondern die `AddLine`-Methode, die Sie auf das `GraphicsPath`-Objekt beziehen. Für andere `Draw`-Methoden gibt es ähnliche

AddXXX-Äquivalente. Auf diese Weise erstellen Sie den Linienzug zunächst, ohne ihn konkret zu zeichnen.

Haben Sie den Linienzug komplett aufgebaut, entscheiden Sie sich, ob Sie die Anweisung

```
locLinienverbund.CloseFigure()
```

anwenden, die den letzten Punkt des Linienzugs automatisch mit dem Anfangspunkt des Linienzugs verbände (für unser Beispiel nicht erwünscht).

Das eigentliche Zeichnen des Linienzugs passiert erst jetzt mit der Anweisung

```
g.DrawPath(locPen, locLinienverbund)
```

So können Sie die Linienzüge schließen, die wirklich geschlossen werden sollen. Ein Doppelklick in das Formular demonstriert diese Vorgehensweise. Das Ergebnis sehen Sie in Abbildung 29.5:

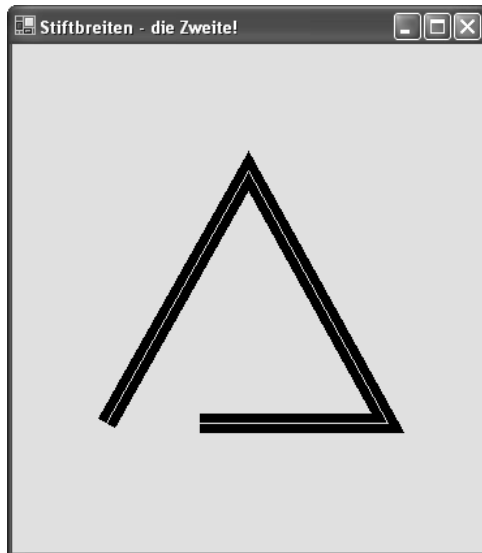


Abbildung 29.5: Mit dem *GraphicsPath*-Objekt erstellen Sie Linienzüge, bei denen die einzelnen Komponenten wirklich miteinander verbunden sind

Der Code, der beide Figuren in das Formular zaubert, sieht dabei auszugsweise folgendermaßen aus:

```
'Zeichnet jeweils ein Rechteck in einer dicken, schwarzen und einer  
'dünnen, gelben Umrandung.  
Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)  
    'Nur um Tipparbeit zu sparen,  
    Dim g As Graphics = e.Graphics  
    'Hier werden die Eckpunkte des offenen Dreiecks gespeichert,  
    Dim locDreiecksPunkte(3) As Point  
    'Diese Variable dient nur dem Sparen der Tipparbeit,  
    Dim locCB As Size = Me.ClientSize  
    'Hier werden die Eckpunkte des Dreiecks definiert,  
    Dim c As Integer  
    'Zählvariable für die Schleife zum Zeichnen der Linien  
    locDreiecksPunkte(0) = New Point(locCB.Width \ 5, (locCB.Height \ 4) * 3)  
    locDreiecksPunkte(1) = New Point(locCB.Width \ 2, locCB.Height \ 4)  
    locDreiecksPunkte(2) = New Point((locCB.Width \ 5) * 4, (locCB.Height \ 4) * 3)
```

```

locDreiecksPunkte(3) = New Point((locCB.Width \ 5) * 2, (locCB.Height \ 4) * 3)

'Zwei mögliche Malverfahren. Das ändert sich bei jedem Doppelklick:
If Not myDoppelklickTrigger Then
    'Einzelne Linien malen.
    Dim locPen As New Pen(Color.Black, 15)
    'Alle Eckpunkte per Linie miteinander verbinden.
    For c = 0 To locDreiecksPunkte.Length - 2
        g.DrawLine(locPen, locDreiecksPunkte(c).X, locDreiecksPunkte(c).Y, _
            locDreiecksPunkte(c + 1).X, locDreiecksPunkte(c + 1).Y)
    Next

    'Das Gleiche nochmal in gelb und dünn.
    locPen = New Pen(Color.Yellow, 1)
    For c = 0 To locDreiecksPunkte.Length - 2
        g.DrawLine(locPen, locDreiecksPunkte(c).X, locDreiecksPunkte(c).Y, _
            locDreiecksPunkte(c + 1).X, locDreiecksPunkte(c + 1).Y)
    Next

Else
    '...oder als Linienzug mithilfe des GraphicPaths-Objektes:
    Dim locPen As New Pen(Color.Black, 15)
    Dim locLinienverbund As New GraphicsPath
    For c = 0 To locDreiecksPunkte.Length - 2
        locLinienverbund.AddLine(locDreiecksPunkte(c).X, locDreiecksPunkte(c).Y, _
            locDreiecksPunkte(c + 1).X, locDreiecksPunkte(c + 1).Y)
    Next

    'Mit dieser Anweisung würden Sie den Endpunkt des Linienzuges
    'mit dem Startpunkt verbinden und so die Figur schließen.
    locFigur.CloseFigure()
    e.Graphics.DrawPath(locPen, locLinienverbund)
    locPen = New Pen(Color.Yellow, 1)
    e.Graphics.DrawPath(locPen, locLinienverbund)
    e.Graphics.DrawPath(locPen, locLinienverbund)
End If

End Sub

Protected Overrides Sub OnDoubleClick(ByVal e As System.EventArgs)
    myDoppelklickTrigger = Not myDoppelklickTrigger
    Invalidate()
End Sub

```

HINWEIS: Wenn Sie ein GraphicsPath-Objekt erstellen, dann ist dieses nicht auf nur eine Figur beschränkt. Mit Hilfe der Methode StartFigure können Sie innerhalb desselben GraphicPaths einen neuen Linienzug beginnen, ohne dass der alte zunächst geschlossen wird. CloseFigure hingegen schließt den aktuellen Linienzug (verknüpft also den letzten mit dem ersten Punkt). Einige Methoden, wie beispielsweise AddEllipse, fügen dem GraphicsPath einen geschlossenen Linienzug direkt hinzu.

Zeichnen des Gitters:

```
Private Sub paintGrid(ByVal g As Graphics)

    Dim locPen As New Pen(GridColor, GridLineWidth)
    Dim locBrush As New SolidBrush(GridColor)

    If GridStyle = GridStyle.Dots Then
        For x As Single = 0 To ClientSize.Width Step GridSize.Width
            For y As Single = 0 To ClientSize.Height Step GridSize.Height
                g.FillRectangle(locBrush, x, y, GridLineWidth, GridLineWidth)
            Next
        Next

    Else:
        'Alte Version
        'For x As Single = 0 To ClientSize.Width Step GridSize.Width
        '    g.DrawLine(locPen, x, 0, x, ClientSize.Height)
        'Next

        'For y As Single = 0 To ClientSize.Height Step GridSize.Height
        '    g.DrawLine(locPen, 0, y, ClientSize.Width, y)
        'Next

        'Aufgepasst: Dieses Konstrukt funktioniert nicht, wegen Rundungsfehlern!
        'Dim locStart, locEnd As Single
        'locStart = GridLineWidth / 2
        'locEnd = ClientSize.Width

        'For x As Single = locStart To locEnd Step GridSize.Width
        '    Problem: locEnd würde niemals genau gleich x sein, wg. Rundungsfehler,
        '    die letzte Linie damit nie an die richtige Stelle gemalt.
        '    If x = locStart Or x = locEnd Then
        '        g.DrawLine(locPen, x, GridLineWidth / 2, x, ClientSize.Height)
        '    Else
        '        g.DrawLine(locPen, x - GridLineWidth / 2, GridLineWidth / 2, x - GridLineWidth / 2, _
        '            ClientSize.Height)
        '    End If
        'Next

        'locStart = GridLineWidth / 2
        'locEnd = ClientSize.Height
        'For y As Single = locStart To locEnd Step GridSize.Height
        '    If y = locStart Or y = locEnd Then
        '        g.DrawLine(locPen, GridLineWidth / 2, y, ClientSize.Width, y)
        '    Else
        '        g.DrawLine(locPen, GridLineWidth / 2, y - GridLineWidth / 2, _
        '            ClientSize.Width, y - GridLineWidth / 2)
        '    End If
        'Next
    End If
End Sub
```

Zu diesem, auskommentierten Teil des Listings vielleicht noch ein paar Anmerkungen, damit Sie diese typischen Fehler beim Arbeiten mit Grafikkoordinaten, die Sie mit *Single*-Werten bestimmen, in Projekten von vornherein vermeiden können. Hier startete der Programmierer den Versuch, das Zeichnen der jeweils ersten und letzten Linie des Gitters durch Koordinatenvergleich zu entdecken – doch dieser Vorgang ist bei vielen *Single*-Werten grundsätzlich zum Scheitern verurteilt: Beim Vergleichen der aktuell verarbeiteten Koordinate mit dem bekannten Wert der jeweils letzten Koordinate durch den Gleichheitsoperator wird der Ausdruck

```
x = locEnd
```

bzw.

```
y = locEnd
```

niemals wahr. Zwar entspricht *locEnd* dem Wert *x* bzw. *y* *fast*, durch die Umwandlung vom Dezimal- in das Binärsystem und die sich daraus kumulierenden Rundungsfehler aber eben nur *fast*. Das bedeutet:

HINWEIS: Vermeiden Sie es grundsätzlich, Programmzustände auf Grund von Fließkommavergleichen festzustellen. Sie können mit *Double*- bzw. *Single*-Werten gefahrlos rechnen und die errechneten Ergebnisse zur Bestimmung von Koordinaten verwenden. Durch kumulierte Rundungsfehler schlagen Vergleiche aber in den meisten Fällen fehl, und Ihr Programm arbeitet nicht wie erwartet. Im hier gezeigten Beispiel könnte *locEnd* beispielsweise den Wert 477,5 innehaben; *x* ist durch kumulierte Rundungsfehler im entscheidenden Moment aber nicht 477,5, sondern 477,500001 – und das ist zwar *fast* 477,5 aber nicht ausreichend, um im Vergleichsausdruck *True* zurückzuliefern und die Sonderbehandlung für die als zuletzt gemalt erkannte Linie einzuleiten.

```
Dim locStart, locEnd As Single
locStart = GridLineWidth / 2
locEnd = ClientSize.Width - GridSize.Width

'Erste Linie Sonderfall:
g.DrawLine(locPen, GridLineWidth / 2, 0, GridLineWidth / 2, ClientSize.Height)

'Mittlere Linien malen.
For x As Single = locStart To locEnd Step GridSize.Width
    g.DrawLine(locPen, x - GridLineWidth / 2, GridLineWidth / 2, _
        x - GridLineWidth / 2, ClientSize.Height)
Next

'Letzte Linie Sonderfall.
g.DrawLine(locPen, ClientSize.Width - GridLineWidth / 2, 0, _
    ClientSize.Width - GridLineWidth / 2, ClientSize.Height)

'Horizontale Linien:
locStart = GridLineWidth / 2
locEnd = ClientSize.Height - GridSize.Height

'Erste Linie Sonderfall:
g.DrawLine(locPen, 0, GridLineWidth / 2, _
    ClientSize.Width, GridLineWidth / 2)
```

```
For y As Single = locStart To locEnd Step GridSize.Height
    g.DrawLine(locPen, GridLineWidth / 2, y - GridLineWidth / 2, _
        ClientSize.Width, y - GridLineWidth / 2)
Next

'Letzte Linie Sonderfall:
g.DrawLine(locPen, 0, ClientSize.Height - GridLineWidth / 2, _
    ClientSize.Width - GridLineWidth / 2, ClientSize.Height - GridLineWidth / 2)
```

End If

End Sub

Stattdessen machen Sie es besser wie in dieser Version der *Gitter-Mal*-Prozedur gezeigt. Die Behandlung der Sonderfälle ist absolut codiert und ihre Ausführung obliegt keiner bedingten Programmverzweigung.