

Kapitel 14

Grafische Grundelemente

In diesem Kapitel:

Grundlagen	306
Die Grafik-Auflösung	313
Die grafischen Grundelemente	315
Zusammenfassung	329

Ohne die grafischen Grundelemente wie Linien, Rechtecke, Ellipsen u.v.m. geht in einem fensterorientierten System eigentlich gar nichts. WPF hat einiges in diesem Bereich zu bieten, denn das *P* in WPF steht ja bekanntlich für *Presentation*. In diesem Kapitel möchte ich Ihnen die Möglichkeiten vorstellen, welche Sie mit diesen Grafikelementen haben. In der zweiten Hälfte des Kapitels werden wir uns mit den komplexen Grafikelementen beschäftigen, dazu gehören die Pfade, Polygone und Bezierkurven.

BEGLEITDATEIEN

Unter `.\Samples\Chapter14` finden Sie die Beispieldateien für dieses Kapitel.

Grundlagen

Grafische Grundelemente können mit WPF überall in der Benutzerschnittstelle verwendet werden. Eine Trennung von Steuerelementen und grafischen Elementen ist hier nicht vorhanden. Grafische Elemente, wie Linien und Ellipsen, sind genau so in einer Benutzerschnittstelle einsetzbar, wie Schaltflächen oder Textboxen. Auch das Programmierparadigma ist identisch, wie das Beispiel in Listing 14.1 zeigt.

```
<Window x:Class="TextKomplex.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Text mit Grafik" Height="150" Width="650"
  >
  <DockPanel>
    <StackPanel Height="30" DockPanel.Dock="Top" Orientation="Horizontal">
      <TextBlock FontSize="20">Mit WPF</TextBlock>
      <Ellipse Width="20" Fill="Red" />
      <TextBlock VerticalAlignment="Center">kann man Grafik</TextBlock>
      <Rectangle Width="30" Fill="Blue" />
      <TextBlock VerticalAlignment="Center">einfach mischen. Und einen</TextBlock>
      <Button Width="120">
        <StackPanel Orientation="Horizontal">
          <Ellipse Width="30" Height="10" Stroke="Blue" Fill="Green" />
          <TextBlock>Hallo</TextBlock>
          <Ellipse Width="30" Height="10" Stroke="Blue" Fill="Red" />
        </StackPanel>
      </Button>
      <TextBlock VerticalAlignment="Center">kann man auch einfügen!</TextBlock>
      <Ellipse Width="30" Fill="Yellow" Stroke="Black" />
    </StackPanel>
  </DockPanel>
</Window>
```

Listing 14.1 Texte und grafische Elemente

Im Beispiel aus Listing 14.1 wird in einem `DockPanel` ein `StackPanel` mit horizontaler Orientierung platziert. Im `StackPanel` können nun sowohl Texte (`TextBlock`) als auch grafische Elemente (`Ellipse`, `Rectangle`) in beliebiger Anordnung dargestellt werden. Auch Steuerelemente, wie hier eine Schaltfläche (`Button`), können eingesetzt werden. Die Schaltfläche selbst kann dann wieder aus grafischen Elementen und Texten bestehen. Das Ergebnis kann in Abbildung 14.1 betrachtet werden.

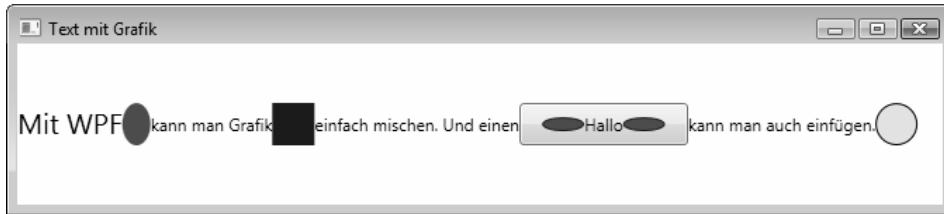


Abbildung 14.1 Texte und Grafik gemischt

Die grafischen Ausgaben können allerdings nicht nur in XAML definiert, sondern auch durch normalen Programmcode erzeugt werden. Hierbei werden Sie jedoch schnell feststellen, dass eine hierarchische Darstellung der einzelnen Elemente in XAML meistens viel übersichtlicher und einfacher zu lesen ist. In Listing 14.2 wird das Fenster aus Abbildung 14.1 nur durch VB-Code erzeugt.

```
Imports System
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Media
Imports System.Windows.Shapes

Namespace TextKomplex
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
            InitMyLayout()
        End Sub

        Private Sub InitMyLayout()
            Dim dock As New DockPanel

            Dim stack As New StackPanel
            stack.Orientation = Orientation.Horizontal
            stack.Height = 30

            Dim t1 As New TextBlock
            t1.FontSize = 20
            t1.Text = "Mit WPF"
            stack.Children.Add(t1)

            Dim e1 As New Ellipse
            e1.Width = 20
            e1.Fill = Brushes.Red
            stack.Children.Add(e1)

            Dim t2 As New TextBlock
            t2.Text = "kann man Grafik"
            t2.VerticalAlignment = Windows.VerticalAlignment.Center
            stack.Children.Add(t2)
```

```
Dim r1 As New Rectangle
r1.Width = 30
r1.Fill = Brushes.Blue
stack.Children.Add(r1)

Dim t3 As New TextBlock
t3.Text = "einfach mischen. Und einen"
t3.VerticalAlignment = Windows.VerticalAlignment.Center
stack.Children.Add(t3)

Dim btn As New Button
btn.Width = 120

Dim pa As New StackPanel
pa.Orientation = Orientation.Horizontal

Dim ebtn1 As New Ellipse
ebtn1.Width = 30
ebtn1.Height = 10
ebtn1.Fill = Brushes.Green
ebtn1.Stroke = Brushes.Blue
pa.Children.Add(ebtn1)

Dim tbtn As New TextBlock
tbtn.Text = "Hallo"
pa.Children.Add(tbtn)

Dim ebtn2 As New Ellipse
ebtn2.Width = 30
ebtn2.Height = 10
ebtn2.Fill = Brushes.Red
ebtn2.Stroke = Brushes.Blue
pa.Children.Add(ebtn2)
btn.Content = pa

stack.Children.Add(btn)

Dim t4 As New TextBlock
t4.Text = "kann man auch einfügen."
t4.VerticalAlignment = Windows.VerticalAlignment.Center
stack.Children.Add(t4)

Dim e2 As New Ellipse
e2.Width = 30
e2.Fill = Brushes.Yellow
e2.Stroke = Brushes.Black
stack.Children.Add(e2)

dock.Children.Add(stack)
Me.Content = dock
End Sub
End Class
End Namespace
```

Listing 14.2 Definition von Hierarchien mit VB-Code ist oft aufwendig

Die Darstellung von grafischen Elementen auf der Schaltfläche im obigen Beispiel ist ebenfalls sehr einfach. Hierbei sollten Sie beachten, dass WPF sehr flexibel ist, was die Hierarchie der einzelnen Elemente angeht. So ist es z.B. ohne weiteres möglich, ein Grid-Element auf einer Schaltfläche zu platzieren, um dann die einzelnen Zellen zur Darstellung von grafischen Elementen auszunutzen. Ein grafisches Element kann auch mehrere Zellen im Grid belegen. Listing 14.3 zeigt ein Beispiel.

```
<Window x:Class="ButtonKomplex.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Eine Super-Schaltfläche" Height="300" Width="400" Background="LightGray"
  >
<Button Width="250" Height="150">
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition Width="3*" />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition Height="3*" />
      <RowDefinition />
    </Grid.RowDefinitions>

    <Rectangle Name="rect1" Grid.Column="0" Grid.Row="0" Fill="Red" Stroke="Black" />
    <Rectangle Name="rect2" Grid.Column="2" Grid.Row="2" Fill="Red" Stroke="Black" />

    <TextBlock Name="text1" Grid.Column="2" Grid.Row="0"
      FontSize="14" VerticalAlignment="Top" Text="Klicken!" />

    <TextBlock Name="text2" Grid.Column="0" Grid.Row="2"
      FontSize="14" VerticalAlignment="Bottom" Text="Klicken!" />

    <Image Grid.Column="1" Grid.Row="1"
      Source="D:\Avalon-Buch\Buch-Dateien\Kap04\Astro.jpg" />

    <Ellipse Name="ellip" Grid.Column="0" Grid.Row="0"
      Grid.ColumnSpan="3" Grid.RowSpan="3"
      Width="230" Height="130" Stroke="Blue" StrokeThickness="5" />
  </Grid>
</Button>
</Window>
```

Listing 14.3 Eine Schaltfläche mit grafischen Elementen



Abbildung 14.2 Die grafische Schaltfläche aus Listing 14.3

Die Schaltfläche in Abbildung 14.2 wird nicht über eine Bitmap, also eine pixelorientierte Grafik realisiert, sondern besteht weiter aus den einzelnen vektororientierten, grafischen Grundelementen, die z.B. auch auf Ereignisse reagieren können.

Bei vielen Benutzeroberflächen-Technologien müssen Steuerelemente von Grund auf neu programmiert werden, wenn sich die Darstellung ändert. Oft ist das Verhalten beim Neuzeichnen dieser Steuerelemente nicht effizient und sie werden öfter gezeichnet, als eigentlich notwendig ist. Dies führt manchmal zu »flackernden« Benutzeroberflächen, mit denen das Arbeiten nicht sehr angenehm ist.

WPF geht hier einen anderen Weg. Die grafischen Elemente werden ebenso wie die Steuerelemente in einer baumähnlichen Struktur verwaltet. Grafiken können dadurch genauso wie z.B. eine Schaltfläche durch Änderung ihrer Eigenschaften modifiziert werden. Der Programmierer muss sich jedoch nicht um das korrekte Neuzeichnen des Fensters oder des Fensterausschnitts kümmern.

Das Beispiel aus Listing 14.3 soll so erweitert werden, dass die Schaltfläche das Click-Ereignis bedient:

```
<Button Width="250" Height="150" Click="OnClick">
```

Die Elemente, die aus dem VB-Code heraus verändert werden sollen, sind in Listing 14.3 bereits mit den Objektnamen versehen. Es muss also nur noch die Methode für das Click-Ereignis implementiert werden (Listing 14.4). Beim Anklicken der Schaltfläche ändert sich sofort das Aussehen derselben. Es wird keine Methode zum Neuzeichnen der Grafiken aufgerufen.

```
Imports System
Imports System.Windows

Namespace ButtonKomplex
    Partial Public Class Window1
        Inherits System.Windows.Window

        Public Sub New()
            InitializeComponent()
        End Sub
    End Class
End Namespace
```

```

Private Sub OnClicked(ByVal sender As Object, ByVal e As RoutedEventArgs)
    rect1.Fill = Brushes.Yellow
    rect2.Fill = Brushes.Yellow
    text2.FontStyle = FontStyles.Italic
    ellip.Stroke = Brushes.Red
    ellip.StrokeThickness = 8

    Dim d As New DoubleCollection
    d.Add(4)
    d.Add(1)
    d.Add(3)
    d.Add(3)
    ellip.StrokeDashArray = d
End Sub
End Class
End Namespace

```

Listing 14.4 Veränderung der Objekte aus VB-Code

Im nächsten Beispiel soll geprüft werden, wie die Behandlung der Ereignisse solcher Grafikelemente funktioniert. In einem Fenster werden mit XAML mehrere Rechtecke (Rectangle) erzeugt. Jedes Rechteck beinhaltet eine Methode für die Verarbeitung des Click-Ereignisses. Beim Anklicken wird das jeweilige Rechteck um 10 Einheiten vergrößert. Hierbei stellen sich nun drei interessante Fragen:

- Vergrößert sich der Bereich für das Click-Ereignis entsprechend der Vergrößerung des Rechtecks?
- Wie kann man das angeklickte Rechteck identifizieren?
- Was passiert, wenn Rechtecke angeklickt werden, die so groß sind, dass sie übereinander liegen?

Listing 14.5 zeigt den entsprechenden VB- und XAML-Code und Abbildung 14.3 zeigt eine mögliche Ausgabe im Fenster.

```

<!-- XAML: Window1.xaml -->
<Window x:Class="TestEreignis.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Test Ereignisse" Height="300" Width="500"
    >
    <Canvas>
        <Rectangle Canvas.Left="10" Canvas.Top="30" Fill="Blue"
            Width="40" Height="40" MouseLeftButtonDown="OnClicked" />
        <Rectangle Canvas.Left="110" Canvas.Top="30" Fill="Red"
            Width="40" Height="40" MouseLeftButtonDown="OnClicked" />
        <Rectangle Canvas.Left="210" Canvas.Top="30" Fill="Black"
            Width="40" Height="40" MouseLeftButtonDown="OnClicked" />
        <Rectangle Canvas.Left="310" Canvas.Top="30" Fill="Green"
            Width="40" Height="40" MouseLeftButtonDown="OnClicked" />
        <Rectangle Canvas.Left="410" Canvas.Top="30" Fill="Yellow"
            Width="40" Height="40" MouseLeftButtonDown="OnClicked" />
    </Canvas>
</Window>

```

```

Imports System
Imports System.Windows

Namespace TestEreignis
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnClicked(ByVal sender As Object, ByVal e As RoutedEventArgs)
            Dim r As Rectangle = CType(sender, Rectangle)
            r.Width += 10
            r.Height += 10
        End Sub
    End Class
End Namespace

```

Listing 14.5 Mehrere Grafikelemente mit einer Ereignismethode

In der Ereignismethode können Sie (wie z.B. aus Windows Forms bekannt) einfach das Objekt mit dem Namen `sender` in ein `Rectangle`-Objekt konvertieren. Danach können Sie, wie gewohnt, auf die Eigenschaften des Elements zugreifen und diese ändern. Im Beispiel wird nicht explizit eine Neuzeichnungsmethode aufgerufen. WPF zeichnet das Element nach der Änderung der Eigenschaften `Width` und `Height` neu. Mit der Vergrößerung des Rechtecks erweitert sich auch der Bereich für das `MouseDown`-Ereignis ohne weiteren Programmieraufwand.

Wenn die einzelnen Rechtecke größer werden und sich überlagern, so erfolgt die Darstellung in der Reihenfolge der Definition aus dem XAML-Code. Zuerst wird also das blaue Rechteck, dann das rote und zuletzt das gelbe Rechteck gezeichnet.

Beim Experimentieren mit dem Beispiel können Sie feststellen, dass bei überlagerten Rechtecken nur das `Click`-Ereignis für das jeweils ganz oben liegende Rechteck ausgelöst wird. Das `Click`-Ereignis wird nicht an die eventuell (optisch) darunter liegenden Rechtecke weitergeleitet, da alle Rechtecke in der Element-Hierarchie (logischer Baum) in der gleichen Ebene liegen.

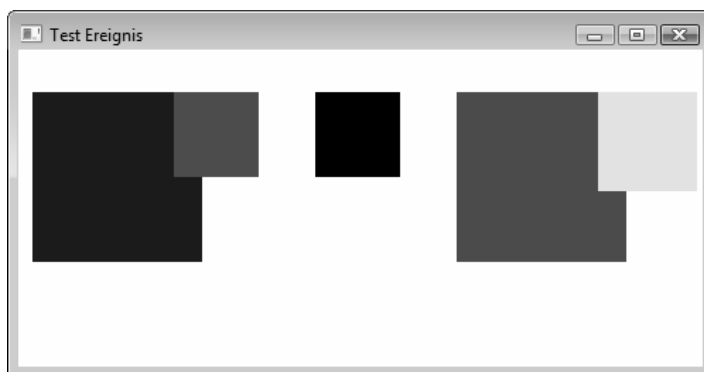


Abbildung 14.3 Alle Rechtecke werden in einer Ereignismethode behandelt

Die Grafik-Auflösung

Unsere Rechner sind in den letzten Jahren nicht nur schneller geworden, auch die Grafikkarten sind im Laufe der Zeit immer weiter verbessert worden. Mein erster Computer, ein Radio Shack TRS-80 Level II, hatte einen Z80-Prozessor mit 0,9 MHz Taktfrequenz. Die Grafikauflösung auf einem Schwarz/Weiß-Monitor betrug sagenhafte 128x48 »Klötze« (Nein, da fehlt keine Null hinten!). Aber auch bei diesen Auflösungen konnten verschiedene Programme schon 3D-Darstellungen von einfachen Objekten auf den Bildschirm zaubern. Eigentlich kaum vorstellbar.

In der heutigen Zeit sind die Grafikkarten wesentlich leistungsfähiger. Aber nicht jede Software nutzt alle Möglichkeiten der Grafikhardware aus. Grafiken, die in beliebiger Weise vergrößert- oder verkleinert werden können, kann man mit Windows schon seit langer Zeit programmieren. Die Grundlagen hierfür finden Sie in der GDI.DLL von Windows. Diese Grafik-Bibliothek ist im Jahr 2001 durch GDI+ noch wesentlich erweitert worden. Trotzdem ist es nicht möglich, mit vertretbarem Aufwand und minimalen Einschränkungen eine vollständig skalierbare Benutzerschnittstelle (hier ist auch die Größe der Steuerelemente gemeint) für Windows-Anwendungen zu erstellen.

Wie bereits weiter oben erwähnt, gibt es diese Trennung von Grafik- und Benutzerschnittstellenelementen in WPF nicht mehr. Eine in der Größe änderbare Benutzerschnittstelle sollte also möglich sein. Wie sieht es aber mit der Darstellungsqualität aus? Wenn Rastergrafiken (Bitmaps) vergrößert werden, erscheinen sehr schnell die unansehnlichen »Klötze« auf dem Bildschirm, die das Betrachten der Grafiken eher zu einem unangenehmen Erlebnis machen. Aber auch hier hat sich in WPF Einiges getan! Alle Elemente der Benutzerschnittstelle werden aus den vektororientierten Grafikelementen erstellt. Es werden keine Rastergrafiken verwendet.

Im folgenden Beispiel (Listing 14.6) können Sie eine Schaltfläche sehen, die mit einer `LayoutTransform` in unterschiedliche Größen transformiert wurde. Die Schaltfläche bleibt hierbei immer exakt die gleiche, nur die Größe wird durch *Zoomen* geändert. Die hierfür verwendete Operation ist eine `LayoutTransform`. Diese Art der Transformation werden wir im Laufe dieses Kapitels noch genauer kennen lernen. Im Moment genügt es, wenn Sie wissen, dass diese Transformation die gesamte Benutzerschnittstelle beeinflussen kann. Dabei muss die Transformation nicht für jedes Element neu angegeben werden. Im Beispiel wird die `LayoutTransformation` nur für die Schaltfläche definiert. Alle Elemente, die in der Hierarchie unter der Schaltfläche liegen (`Ellipse`, `TextBlock`, `Rectangle` und `Line`) werden automatisch entsprechend skaliert.

```
<Window x:Class="ButtonGross.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Schaltfläche transformieren" Height="300" Width="600"
  >
  <StackPanel>
    <Button Height="50" Width="250" Margin="10">
      <Button.LayoutTransform>
        <ScaleTransform ScaleX="1" ScaleY="1" />
      </Button.LayoutTransform>

      <StackPanel Orientation="Horizontal">
        <Ellipse Width="50" Height="25" Fill="Red" Stroke="Black" />
        <TextBlock FontSize="20">Klicken!</TextBlock>
      </StackPanel>
    </Button>
  </StackPanel>
</Window>
```

```

<Rectangle Width="50" Height="25" Stroke="Blue" RadiusX="8" RadiusY="8" />
<Line X1="0" Y1="0" X2="50" Y2="25" Stroke="Green" StrokeThickness="3" />
</StackPanel>
</Button>
</StackPanel>
</Window>

```

Listing 14.6 Eine Schaltfläche mit einer LayoutTransformation

In Abbildung 14.4 werden mehrere der Schaltflächen dargestellt. Hierbei wurden die Skalierungsfaktoren (`ScaleX` und `ScaleY`) für die `LayoutTransform` auf die Werte 1, 2, 3 und 5 eingestellt. Die Schaltfläche wird Schritt für Schritt größer. Trotzdem bleibt die Qualität der Grafik sehr hoch. Das Bild wird nicht unscharf und enthält keine »Klötze«. Alle Grafikelemente, die Schaltfläche und der Text werden nicht als Rastergrafik vergrößert, sondern mit dem jeweiligen Zoomfaktor neu gezeichnet. Dabei werden auch die Linienstärken der einzelnen Grafikelemente entsprechend vergrößert, sodass die gesamte Darstellung immer noch »ausgewogen« aussieht.

Genauso, wie die Schaltfläche in Abbildung 14.4 skaliert wurde, können Sie jedes Element der Benutzeroberfläche vergrößern oder verkleinern. Aber damit noch nicht genug. Ebenso einfach können Sie ein Steuerelement oder ein Grafikelement drehen, kippen oder nur an eine andere Stelle verschieben. Auch diese Operationen können mit einer `LayoutTransform` durchgeführt werden.

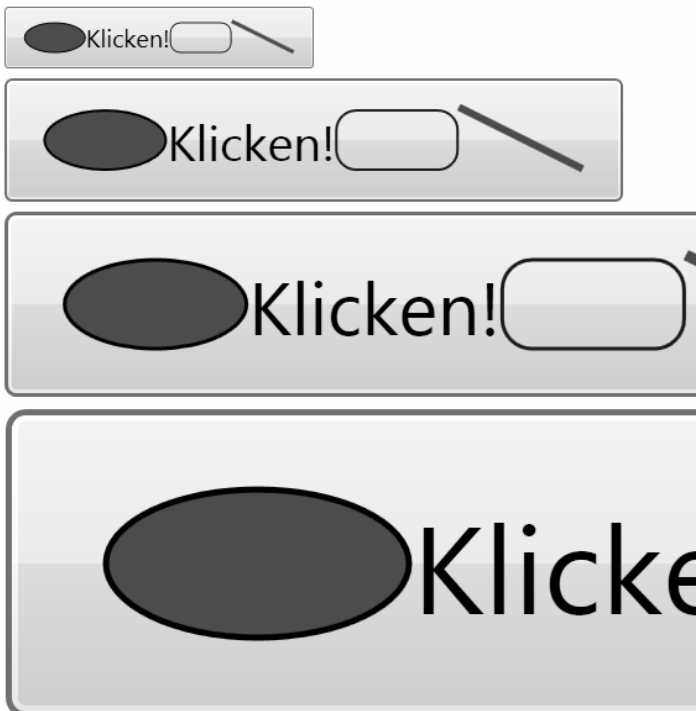


Abbildung 14.4 Die vergrößerte Schaltfläche

Da Sie alle Elemente in einer beliebigen Größe darstellen können, gibt es im Grunde genommen keine feste Beziehung zwischen den Koordinaten der Grafikobjekte und den physikalischen Pixeln des Bildschirms. Ganz abgesehen davon können alle Koordinaten als Fließkommazahlen angegeben werden. In WPF werden die Koordinaten in so genannten *geräteunabhängigen Pixeln* angegeben. Ein geräteunabhängiger Pixel entspricht 1/96 Zoll. Anders ausgedrückt entsprechen 96 Pixel einer Länge von 25,4 mm auf dem Bildschirm. Die Anzahl der tatsächlichen, physikalischen Pixel, die hierzu benötigt werden, hängt von der Auflösung des Bildschirms ab.

Der »krumme« Wert von 96 DPI (Dotch per Inch) pro Zoll kommt daher, dass in Windows die Auflösung des Standard-Displays 96 DPI ist. Für das Standard-Display gilt somit: 1 logischer Pixel = 1 echter Pixel! Für andere Wiedergabegeräte werden alle Koordinaten entsprechend umgerechnet.

Die grafischen Grundelemente

Beginnen wir mit dem Zusammenspiel der grafischen Grundelemente (Shapes), der Stifte (Pens) und der Pinsel (Brushes). Es gibt diverse Klassen, welche die grafischen Grundelemente zur Verfügung stellen. Diese Klassen beinhalten verschiedene Eigenschaften, um das Aussehen zu beeinflussen.

- Rectangle
- Ellipse
- Line
- Polyline
- Polygon
- Path

Für diese grafischen Objekte kann man Füllfarben in Form von Pinseln (Brush) auswählen. Der äußere Rand kann durch einen Stift (Pen) definiert werden. Der einfachste Pinsel ist ein `SolidColorBrush`, der nur aus einer vorgegebenen Farbe besteht. Andere Pinsel, die wir später noch kennen lernen werden, können dagegen komplexe Farbverläufe darstellen. Listing 14.7 zeigt, wie Sie Ellipsen, Rechtecke und abgerundete Rechtecke zeichnen können. Die Füllfarbe wird mit der Eigenschaft `Fill` bestimmt, die Randfarbe mit der Eigenschaft `Stroke`. Die Dicke der Randlinie wird mit der Eigenschaft `StrokeThickness` angegeben. Schließlich können Sie beliebige gestrichelte Linien erzeugen. Hierzu geben Sie mit der Eigenschaft `StrokeDashArray` an, wie viele Einheiten des Striches gezeichnet und wie viele Einheiten nicht gezeichnet werden sollen. Sie können hier auch mehrere Zahlen angeben, sodass Sie sehr komplizierte Strichmuster erzeugen können (Abbildung 14.5).

```
<Window x:Class="Shapes1.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Grafische Grundelemente" Height="350" Width="300"
  >
  <Canvas>
    <Ellipse Canvas.Left="10" Canvas.Top="10" Width="80" Height="60" Fill="Red" />
    <Rectangle Canvas.Left="100" Canvas.Top="10" Width="80" Height="60" Fill="Blue" />
    <Rectangle Canvas.Left="190" Canvas.Top="10" Width="80" Height="60"
      RadiusX="15" RadiusY="15" Fill="Green" />
  </Canvas>
</Window>
```

```

<Ellipse Canvas.Left="10" Canvas.Top="80" Width="80" Height="60" Fill="Red" Stroke="Black" />
<Rectangle Canvas.Left="100" Canvas.Top="80" Width="80" Height="60" Fill="Blue" Stroke="Red" />
<Rectangle Canvas.Left="190" Canvas.Top="80" Width="80" Height="60"
    RadiusX="15" RadiusY="15" Fill="Green" Stroke="Aquamarine" />

<Ellipse Canvas.Left="10" Canvas.Top="150" Width="80" Height="60" Fill="Red" Stroke="Black"
    StrokeThickness="3" />
<Rectangle Canvas.Left="100" Canvas.Top="150" Width="80" Height="60" Fill="Blue" Stroke="Red"
    StrokeThickness="7" />
<Rectangle Canvas.Left="190" Canvas.Top="150" Width="80" Height="60" RadiusX="15" RadiusY="15"
    Fill="Green" Stroke="Aquamarine" StrokeThickness="10" />

<Ellipse Canvas.Left="10" Canvas.Top="220" Width="80" Height="60" Fill="Red" Stroke="Black"
    StrokeThickness="3" StrokeDashArray="2 2" />
<Rectangle Canvas.Left="100" Canvas.Top="220" Width="80" Height="60" Fill="Blue" Stroke="Red"
    StrokeThickness="7" StrokeDashArray="2 3 4 2" />
<Rectangle Canvas.Left="190" Canvas.Top="220" Width="80" Height="60" RadiusX="15" RadiusY="15"
    Fill="Green" Stroke="Aquamarine" StrokeThickness="10" StrokeDashArray="1 1"/>
</Canvas>
</Window>

```

Listing 14.7 Verschiedene Ellipsen und Rechtecke

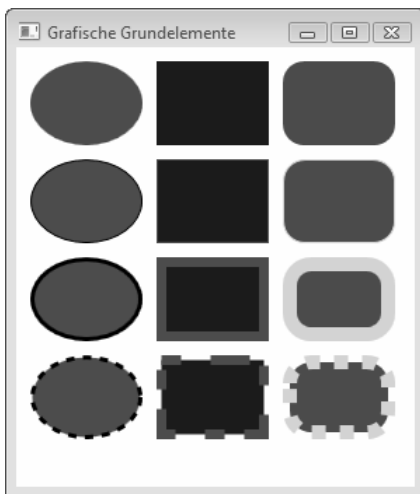


Abbildung 14.5 Verschiedene Ellipsen und Rechtecke

Rechteck und Ellipse

Die grafischen Grundelemente Rectangle, Ellipse, Line, Polygon, Polyline und Path sind von der Basisklasse Shape abgeleitet. Die meisten Eigenschaften der grafischen Grundelemente kommen aus dieser Basisklasse.

Wie Sie bereits gelernt haben, wird der Rand eines grafischen Elements über die Eigenschaft `Stroke` der `Shape`-Klasse angegeben, welche vom Typ `Brush` ist. Intern wird an dieser Stelle ein `Pen`-Objekt benutzt, wie man das eigentlich auch erwarten würde. Damit die XAML-Syntax aber nicht zu kompliziert wird, werden die Eigenschaften des `Pen`-Objektes über eigene Eigenschaften im `Shape`-Objekt abgebildet.

`Rectangle` und `Ellipse` haben wir schon in einem Beispiel kennen gelernt (Listing 14.7). Dort wurde die Positionierung der grafischen Elemente mit einem `Canvas`-Objekt durchgeführt. Die grafischen Elemente können aber auch genauso in einen `DockPanel`, `StackPanel`, `WrapPanel` oder einem `Grid` (Listing 14.8) positioniert werden.

```
<Window x:Class="Shapes2.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Grafik im Grid" Height="200" Width="300"
  >
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="2*" />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="2*" />
      <RowDefinition />
    </Grid.RowDefinitions>

    <Ellipse Grid.Column="0" Grid.Row="0" Fill="Blue" />
    <Rectangle Grid.Column="1" Grid.Row="0" Fill="Red" />
    <Rectangle Grid.Column="0" Grid.Row="1" Fill="Green" />
    <Canvas Grid.Column="1" Grid.Row="1">
      <Ellipse Canvas.Left="10" Canvas.Top="10" Width="70" Height="35"
        Fill="White" Stroke="Black" StrokeThickness="3" />
    </Canvas>
  </Grid>
</Window>
```

Listing 14.8 Die Positionierung von grafischen Elementen mit einem `Grid`

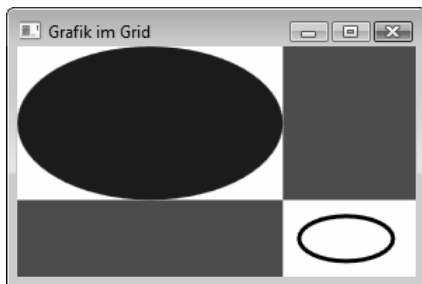


Abbildung 14.6 Grafische Elemente im `Grid`

In Abbildung 14.6 können Sie das Ergebnis des Programms aus Listing 14.8 sehen. In diesem Grid ist die erste Zeile doppelt so hoch wie die zweite Zeile. Dies gilt auch für die Spaltenbreiten. Die blaue Ellipse und die beiden Rechtecke füllen jeweils die gesamte Zelle im Grid aus, da explizit keine Größen (Width, Height) angegeben wurden. Die kleine weiße Ellipse wurde mithilfe eines Canvas-Elementes positioniert. Das Canvas-Element füllt in diesem Fall die gesamte Grid-Zelle aus. Im Canvas selbst kann nun die Ellipse beliebig angeordnet werden.

Einfache Transformationen

Da wir gerade mit Grafiken zu tun haben, stellt sich natürlich auch die Frage, ob es möglich ist, ein schräg liegendes Rechteck oder eine »schräge« Ellipse zu zeichnen. Bisher wurden die Ellipsen und Rechtecke immer so dargestellt, dass die beiden Hauptachsen des Grafikobjektes parallel zu den Achsen des Koordinatensystems lagen. Das soll nun geändert werden.

Die meisten Elemente in WPF enthalten zwei wichtige Transformationen, die als Eigenschaften in den Klassen vorhanden sind: `LayoutTransform` und `RenderTransform`. Die `LayoutTransform`-Eigenschaft wird in erster Linie benutzt, um Teile der Benutzerschnittstelle zu vergrößern oder zu verkleinern. Bei der Anwendung einer `LayoutTransform` werden die einzelnen Elemente in der Benutzerschnittstelle durch das Layout-System von WPF ggf. neu arrangiert. Diese Transformation werden wir später noch genauer kennen lernen.

Die Eigenschaft `RenderTransform` beeinflusst dagegen nur den Inhalt eines Steuerelements. Diese Transformation können wir benutzen, um unsere grafischen Objekte zu drehen. Abbildung 14.7 zeigt das Ergebnis aus Listing 14.9.

```
<Window x:Class="Transform1.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Gedrehte Objekte" Height="350" Width="400"
  >
  <Canvas>
    <Rectangle Canvas.Left="100" Canvas.Top="50" Width="200" Height="50" Fill="Red" />

    <Ellipse Canvas.Left="100" Canvas.Top="50" Width="200" Height="50" Fill="Green">
      <Ellipse.RenderTransform>
        <RotateTransform Angle="30" />
      </Ellipse.RenderTransform>
    </Ellipse>

    <Rectangle Canvas.Left="100" Canvas.Top="50" Width="200" Height="50" Fill="Blue">
      <Rectangle.RenderTransform>
        <RotateTransform Angle="60" />
      </Rectangle.RenderTransform>
    </Rectangle>

    <Ellipse Canvas.Left="100" Canvas.Top="50" Width="200" Height="50" Fill="Violet">
      <Ellipse.RenderTransform>
        <RotateTransform Angle="90" />
      </Ellipse.RenderTransform>
    </Ellipse>
  </Canvas>
</Window>
```

Listing 14.9 Gedrehte Rechtecke und Ellipsen

Bei der Ausführung einer Rotation ist nicht nur der Rotationswinkel von Bedeutung, sondern auch der Punkt, um den die einzelnen Objekte gedreht werden sollen. Im Beispiel aus Listing 14.9 ist das der obere, linke Eckpunkt der Rechtecke (bzw. Ellipsen). Sie können jedoch einen beliebigen Drehpunkt für die Rotation mit den Eigenschaften `CenterX` und `CenterY` im `RotateTransform`-Objekt angeben (Listing 14.10).



Abbildung 14.7 Gedrehte Rechtecke und Ellipsen

Die Transformationen im XAML-Code können folgendermaßen angepasst werden:

```
<Rectangle.RenderTransform>  
  <RotateTransform Angle="60" CenterX="100" CenterY="25" />  
</Rectangle.RenderTransform>
```

Nun liegt der Drehpunkt genau in der Mitte des ersten, roten Rechtecks und die ausgegebene Grafik wird in Abbildung 14.8 gezeigt.



Abbildung 14.8 Der Drehpunkt liegt in der Mitte des blauen Rechtecks

Das Thema »Transformationen« ist hiermit aber noch nicht abgeschlossen. In den Eigenschaften `LayoutTransform` und `RenderTransform` stecken noch viel mehr Möglichkeiten, die dann an den entsprechenden Stellen vorgestellt werden.

Die Linie

Das einfachste grafische Element ist eine Linie (Line). Für eine Linie wird ein Anfangspunkt (X1, Y1) und ein Endpunkt (X2, Y2) definiert. Eine Füllfarbe ist natürlich nicht erforderlich. Mit der Stroke-Eigenschaft wird die Farbe der Linie festgelegt. Die Linienstärke wird mit der Eigenschaft Thickness gesetzt und die Strichelung mit DashArray. Bei Linienobjekten können Sie außerdem das Ende der Linien beeinflussen (Listing 14.10). Diese Möglichkeit ist besonders bei dicken Linien wichtig. In Abbildung 14.9 können Sie erkennen, dass das Linienende über den Anfangs- und Endpunkt der Linie hinausragt.

```
<Window x:Class="Linecap.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Linienenden definieren" Height="160" Width="300"
  >
  <Canvas>
    <Line X1="30" Y1="30" X2="250" Y2="30" Stroke="Black" StrokeThickness="20" />
    <Line X1="30" Y1="60" X2="250" Y2="60" Stroke="Black" StrokeThickness="20"
      StrokeStartLineCap="Round" StrokeEndLineCap="Round" />
    <Line X1="30" Y1="90" X2="250" Y2="90" Stroke="Black" StrokeThickness="20"
      StrokeStartLineCap="Triangle" StrokeEndLineCap="Triangle" />
  </Canvas>
</Window>
```

Listing 14.10 Linienende definieren

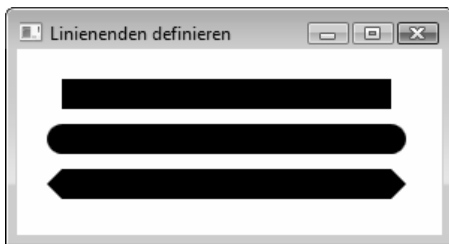


Abbildung 14.9 Verschiedene Linienenden

Die Polylinie

Mit dem Element Polyline können Sie mehrere Linien, die miteinander verbunden sind, zeichnen. Statt für jedes Liniensegment den Anfangs- und den Endpunkt anzugeben, können Sie hier einfach die einzelnen »Eckpunkte« der Gesamtlinie als Zahlenfeld mit der Eigenschaft Points definieren. Der Aufwand ist somit wesentlich geringer, wie auch das Beispiel »Fieberkurve« in Listing 14.11 und Abbildung 14.10 zeigt. Alle anderen, bereits bekannten Eigenschaften können Sie ganz normal benutzen.

```
<Window x:Class="Fieberkurve.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Fieberkurve" Height="180" Width="250"
  >
```

```
<Canvas>
  <Polyline Stroke="Red" StrokeThickness="3"
    Points="10,110 20,50 30,20 60,25 75,20 100,120 115,50 135,60 165,65 200,90" />
</Canvas>
</Window>
```

Listing 14.11 Eine Linie aus vielen Teilstücken

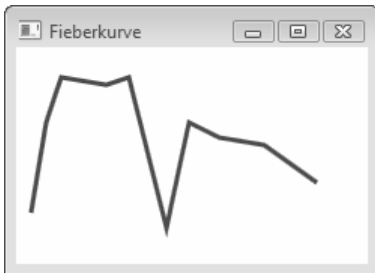


Abbildung 14.10 Ein Polyline-Element als »Fieberkurve«

Das Polygon-Element ist dem Polyline-Element sehr ähnlich. Der einzige Unterschied ist, dass ein Polygon-Element immer geschlossen wird, d.h., der Anfangspunkt der ersten Linie wird mit dem Endpunkt der letzten Linie verbunden. Sie können das letzte Beispiel nehmen, und den XAML-Code geringfügig ändern (Listing 14.12). Das Ergebnis zeigt Abbildung 14.11.

```
<Canvas>
  <Polygon Stroke="Red" StrokeThickness="3"
    Points="10,110 20,50 30,20 60,25 75,20 100,120 115,50 135,60 165,65 200,90" />
</Canvas>
```

Listing 14.12 Eine geschlossene Kurve mit Polygon

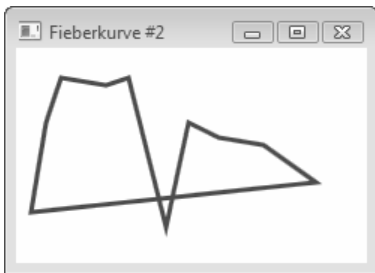


Abbildung 14.11 Geschlossene Kurve mit Polygon

Das automatische Schließen der Kurve vereinfacht viele Zeichenoperationen, da wir uns den Anfangspunkt der gesamten Linie nicht mehr merken müssen. Nun gibt es bei Polygonen noch eine weitere Besonderheit. Da Polygone immer geschlossen werden, kann man sie mit einer Füllfarbe ausfüllen. Aber das ist nicht ganz so einfach, wie es sich im ersten Moment anhört. Wenn wir den Polygon in Abbildung 14.11 betrachten, stellen wir fest, dass es gleich mehrere Flächenteile gibt, die ausgefüllt werden müssen.

Im ersten Versuch wollen wir das letzte Beispiel nur um eine Fill-Eigenschaft erweitern:

```
<Canvas>
  <Polygon Stroke="Red" StrokeThickness="3" Fill="Blue"
    Points="10,110 20,50 30,20 60,25 75,20 100,120 115,50 135,60 165,65 200,90" />
</Canvas>
```

Listing 14.13 Der Polygon soll gefüllt werden

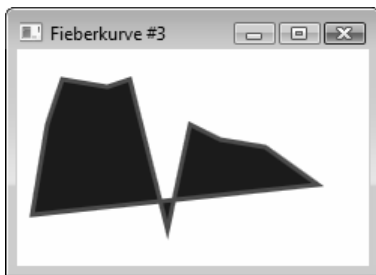


Abbildung 14.12 Polygon mit Fill-Eigenschaft

Die Ausgabe von Listing 14.13 zeigt Abbildung 14.12: Alle Teilflächen des Polygons sind ausgefüllt worden. Für das Füllen von Polygonen gibt es zwei Möglichkeiten, die über die Eigenschaft `FillRule` eingestellt werden. Der Standardwert für diese Eigenschaft ist `EvenOdd`. Das bedeutet, dass nur solche Flächen gefüllt werden, von denen aus eine ungerade Anzahl von Linien bis nach außen überquert werden müssen. Schauen Sie sich einfach das folgende Beispiel in Listing 14.14 an:

```
<Window x:Class="FillRule1.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Füllen" Height="175" Width="150"
  >
  <Canvas>
    <Polygon Stroke="Red" StrokeThickness="6" Fill="Blue" FillRule="EvenOdd"
      Points="20,70 20,20 120,20 120,120 20,120 20,70 40,70 40,40 100,40 100,100 40,100 40,70" />
  </Canvas>
</Window>
```

Listing 14.14 Die Füllregel EvenOdd

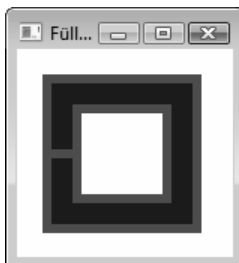


Abbildung 14.13 Die Füllregel EvenOdd

In einem Polygon-Element wurden in Abbildung 14.13 zwei Rechtecke mit der Füllregel EvenOdd ineinander gezeichnet. Dabei wurde das innere Rechteck nicht mit der Füllfarbe blau ausgefüllt. Wenn wir nun eine Hilfslinie aus dem inneren Rechteck nach außen (bis in die Unendlichkeit) legen (Abbildung 14.14, Linie 1), dann werden zwei Linien des Polygons überquert. Da »Zwei« bekanntermaßen eine gerade Zahl ist, wird die innere Fläche nicht gefüllt. Wenn Sie dagegen aus dem Bereich zwischen dem inneren und dem äußeren Rechteck eine Linie bis in die Unendlichkeit zeichnen (Abbildung 14.14, Linie 2 oder 3), werden eine oder drei Polygon-Linien überschritten. Wir haben eine ungerade Zahl, d.h., die Fläche wird mit blauer Farbe ausgefüllt.

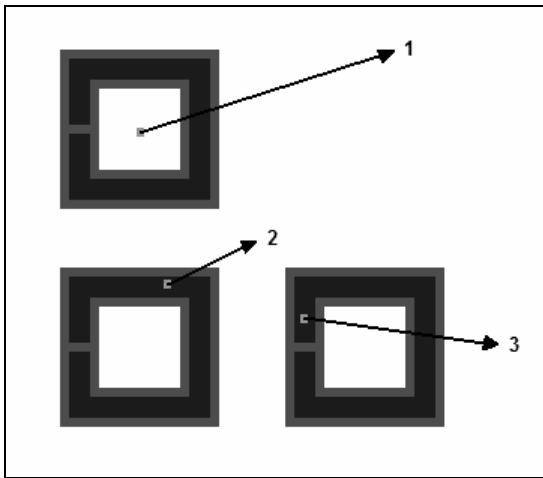


Abbildung 14.14 Die Füllregel EvenOdd

Die zweite Füllregel NonZero ist etwas schwerer verständlich. Für diese Regel benutzen wir ein etwas anderes Beispiel, welches in Listing 14.15 dargestellt ist.

```
<Window x:Class="FillRule2.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Füllen" Height="200" Width="350"
>
<Canvas>
  <Polygon Stroke="Red" StrokeThickness="4" Fill="Blue" FillRule="EvenOdd"
    Points="20,50 20,100 70,100 70,20 130,20 130,70 50,70 50,130 100,130 100,50" />
  <Polygon Stroke="Red" StrokeThickness="4" Fill="Blue" FillRule="NonZero"
    Points="170,50 170,100 220,100 220,20 280,20 280,70 200,70 200,130 250,130 250,50" />
</Canvas>
</Window>
```

Listing 14.15 Die Füllregeln EvenOdd und NonZero

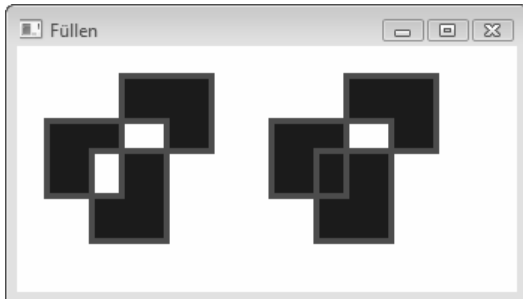


Abbildung 14.15 Die Füllregeln EvenOdd und NonZero

In Abbildung 14.15 sehen Sie auf der linken Seite einen Polygon mit der Füllregel EvenOdd, rechts dagegen ist der gleiche Polygon mit der Regel NonZero dargestellt. Das Ergebnis mit EvenOdd ist klar, wenn wir wieder die Hilfslinien gedacht nach außen ziehen.

Die Füllregel NonZero liefert bei einer ungeraden Anzahl von Linienüberquerungen das gleiche Ergebnis beim Füllen wie die Regel EvenOdd. Wenn jedoch eine gerade Anzahl von Linienüberquerungen notwendig ist, wird die Teilfläche nur dann gefüllt, wenn die Anzahl der Linien, die in eine bestimmte Richtung relativ zur Grafik verlaufen, ungleich der Anzahl der Linien in die andere Richtung sind. Abbildung 14.16 stellt dieses Verhalten etwas genauer dar. Die Zeichenrichtung für die entscheidenden Linien ist dort durch Pfeile dargestellt.

Die Hilfslinie Nr. 1 in Abbildung 14.16 überquert zwei Linien, welche in die gleiche Richtung führen. Da nun in die gegensätzliche Richtung gar keine Linie verläuft, wird der Teilbereich mit der angegebenen Füllfarbe ausgefüllt. Bei der Hilfslinie Nr. 2 werden zwei Linien überquert, von denen eine nach rechts und die andere Linie nach links verläuft. Damit ist die Anzahl der Linien, die jeweils in eine Richtung verläuft, gleich und die Teilfläche wird nicht gefüllt.

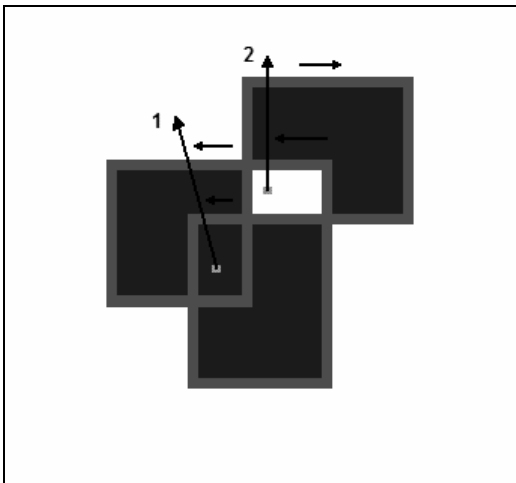


Abbildung 14.16 Die Füllregel NonZero

Das Path-Element

Ein weiteres grafisches Grundelement ist das Path-Element. Es ist das leistungsfähigste 2D-Grafikobjekt in WPF. Mithilfe des Path-Elementes können Sie sehr komplexe Figuren aufbauen und darstellen. Das Path-Element enthält die Eigenschaft `Data`, über die Sie ein oder mehrere grafische Grundelemente definieren können. Die Elemente, die Sie dort benutzen können, heißen `RectangleGeometry`, `EllipseGeometry`, `LineGeometry`, usw. Mehrere Grafikelemente können in einem Path mit einer `GeometryGroup` zusammengefasst werden. Listing 14.16 zeigt ein Beispiel.

```
<Window x:Class="Path1.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ein grafischer Pfad" Height="300" Width="300"
  >
  <Grid>
    <Path Stroke="Red" StrokeThickness="3" Fill="Yellow" >
      <Path.Data>
        <GeometryGroup>
          <RectangleGeometry Rect="0, 0, 150, 100" />
          <EllipseGeometry Center="75, 50" RadiusX="75" RadiusY="50" />
          <LineGeometry StartPoint="0, 0" EndPoint="150,100" />
          <LineGeometry StartPoint="0, 100" EndPoint="150, 0" />
        </GeometryGroup>
      </Path.Data>
    </Path>
  </Grid>
</Window>
```

Listing 14.16 Mit einem Path-Objekt komplexere Grafiken definieren

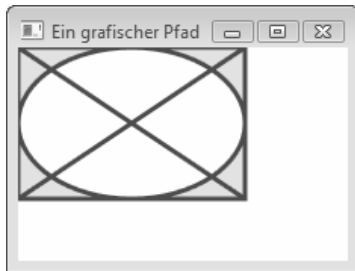


Abbildung 14.17 Das Path-Element mit Ellipse, Rechteck und Linien

Für das Path-Element steht ebenfalls die Eigenschaft `Fill` zur Verfügung, um eine Füllfarbe für die gesamte Figur zu definieren. Im Element `GeometryGroup` dagegen befindet sich die Eigenschaft `FillRule`, die wir aus den vorherigen Beispielen kennen und die sich auch genau so verhält. In **Abbildung 14.17** wird die innere Ellipse nicht gelb ausgefüllt, weil die Eigenschaft `FillRule` standardmäßig auf `EvenOdd` gesetzt ist und die Anzahl der Linien, die von innen nach außen durchquert werden müssen, gerade ist.

Innerhalb eines Path-Elements können Sie sehr komplexe Figuren definieren. In **Listing 14.17** wird eine `PathGeometry` definiert. Diese kann mehrere Figuren beinhalten, die bei Bedarf auch geschlossen werden können. Jede Figur wiederum besteht aus einem Startpunkt und mehreren Segmenten, welche dann die

gewünschte Figur aufbauen. Die Segmente (Tabelle 14.1) können Linien, Bögen oder auch komplexe Kurven sein. Diese einzelnen Segmente werden fortlaufend aneinander gezeichnet. In der Eigenschaft `PathFigure.Segments` können Sie beliebig viele, unterschiedliche Segmente für eine Figur definieren.

```
<Window x:Class="Path2.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Mehrere Teile" Height="300" Width="300"
  >
  <Canvas>
    <Path Fill="Blue" Stroke="Black" StrokeThickness="3">
      <Path.Data>
        <PathGeometry>
          <PathGeometry.Figures>
            <PathFigureCollection>
              <PathFigure StartPoint="50,70" IsClosed="True">
                <PathFigure.Segments>
                  <PathSegmentCollection>
                    <LineSegment Point="100,40" />
                    <LineSegment Point="200,70" />
                    <LineSegment Point="250,100" />
                    <LineSegment Point="250,150" />
                    <ArcSegment Point="200,200" Size="50,50" SweepDirection="Clockwise" />
                  </PathSegmentCollection>
                </PathFigure.Segments>
              </PathFigure>
            </PathFigureCollection>
          </PathGeometry.Figures>
        </PathGeometry>
      </Path.Data>
    </Path>
  </Canvas>
</Window>
```

Listing 14.17 Eine komplexe Figur mit einem Path-Element

Segmenttyp	Verhalten
LineSegment	Einfache gerade Linie
PolyLineSegment	Mehrere gerade Linien
ArcSegment	Bogensegment
BezierSegment	Kubische Bezierkurve
QuadraticBezierSegment	Quadratische Bezierkurve
PolyBezierSegment	Mehrere kubische Bezierkurven
PolyQuadraticBezierSegment	Mehrere quadratische Bezierkurven

Tabelle 14.1 Die möglichen Segmenttypen in einem Path-Element

Hit-Testing mit dem Path-Element

Wenn wir komplexe Grafiken mit dem Path-Element erzeugen, ergibt sich schnell die Frage, ob wir einfach feststellen können, wann in eine geschlossene Kurve hineingeklickt wurde. Eine einfache Demo kann den Sachverhalt klären. Wir erweitern hierzu das Beispiel aus Listing 14.17 und implementieren das `MouseDown`-Ereignis für das Path-Element:

```
<Path Fill="Blue" Stroke="Black" StrokeThickness="3" MouseDown="OnMouseDownPath">
```

Die Ereignismethode wird in VB implementiert und gibt einfach nur eine `MessageBox` aus, wenn das Path-Element mit dem Mausklick getroffen wurde.

```
Imports System
Imports System.Windows
Imports System.Windows.Input

Namespace Path4
    Partial Public Class Window1
        Inherits System.Windows.Window

        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnMouseDownPath(ByVal sender As Object, ByVal e As MouseEventArgs)
            MessageBox.Show("Im Path-Element!")
        End Sub
    End Class
End Namespace
```

Wenn Sie das Beispielprogramm laufen lassen, merken Sie sofort, dass nur Klicks innerhalb des Path-Elements die Meldung erscheinen lassen. Dadurch ist das Hit-Testing in diesem Fall sehr einfach zu implementieren.

Genauso einfach funktioniert das übrigens auch mit `Polygon`-Elementen. Hier werden beim Hit-Testing im `Polygon` die Einstellungen der `FillRule`-Eigenschaft berücksichtigt. Wenn ein Bereich des `Polygon`s mit der gewählten Füllfarbe gezeichnet wird, dann wird auch das `MouseDown`-Ereignis in diesem Bereich ausgelöst (siehe Abbildung 14.15). In den Bereichen, die in Abbildung 14.15 nicht in blau eingefärbt sind, wird auch das `MouseDown`-Ereignis nicht ausgelöst. In den `Polygon`-Elementen wird nur das Mausereignis implementiert, das dann im Code verarbeitet werden kann:

```
<Polygon Stroke="Red" StrokeThickness="4" Fill="Blue" FillRule="EvenOdd" MouseDown="OnMouseDown"
    Points="20,50 20,100 70,100 70,20 130,20 130,70 50,70 50,130 100,130 100,50" />
<Polygon Stroke="Red" StrokeThickness="4" Fill="Blue" FillRule="NonZero" MouseDown="OnMouseDown"
    Points="170,50 170,100 220,100 220,20 280,20 280,70 200,70 200,130 250,130 250,50" />
```

Es gibt jedoch auch komplizierte Fälle beim Hit-Testing, welche die Implementierung von etwas mehr Code erfordern. In den oben gezeigten Beispielen werden nur die Mausklickpositionen für den Hit-Test herangezogen. Wir wollen ein Beispiel nun so abändern, dass ein kreisförmiger Bereich um die Klickposition herum als Testbereich verwendet wird.

Der XAML-Code mit einem Path-Element für das erweiterte Hit-Testing wird in Listing 14.18 gezeigt. Das MouseDown-Ereignis wird in diesem Beispiel im Hauptfenster Window1 »eingehängt«, damit wir das Ereignis auch dann auswerten können, wenn nicht direkt in das Path-Element geklickt wurde. Im Code (Listing 14.19) wird in der Mausereignis-Methode zunächst die Position des Klicks ermittelt. Dann wird ein EllipseGeometry-Objekt angelegt, in dem ein Kreis von 20 Einheiten Durchmesser definiert wird. Nun kann die statische Methode HitTest aus der VisualTreeHelper-Klasse benutzt werden, um im Trefferfall eine Rückrufmethode mit dem Namen HitResult aufzurufen. Dort wird das Hit-Testergebnis ausgewertet. Mit der Eigenschaft IntersectionDetail aus der Klasse GeometryHitTestResult können wir dann ermitteln, ob der angelegte Kreis vollständig, teilweise oder gar nicht im definierten Path-Element liegt. Nachdem ein Treffer erzielt wurde, können Sie die Suche fortführen (HitTestResultBehavior.Continue) oder abbrechen (HitTestResultBehavior.Stop). Eine Steuerung der Suche über die Aufzählung HitTestResultBehavior ist besonders dann interessant, wenn mehrere grafische Objekte übereinander liegen und ermittelt werden soll, ob ein ganz bestimmtes Grafikobjekt angeklickt wurde. Bei der Benutzung von Continue wird die Rückrufmethode für das nächste getroffene Objekt erneut aufgerufen. Dort findet dann die weitere Verarbeitung der Treffer-Informationen statt. Eine Verwendung von Stop beendet die laufende Trefferanalyse und die Rückrufmethode wird nicht mehr aufgerufen, auch wenn es noch weitere Trefferobjekte geben sollte.

```
<Window x:Class="HitTesting.Window1"
  xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
  xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
  Title="Hit-Testing" Height="300" Width="300" MouseDown="OnMouseDown"
  >
  <Grid>
    <Path Name="path" Stroke="Black" StrokeThickness="3" Fill="Red">
      <Path.Data>M 50,50 h 130 l 50,50 v 100 h -100 v -50 h 50 Z</Path.Data>
    </Path>
  </Grid>
</Window>
```

Listing 14.18 Erweitertes Hit-Testing

```
Imports System
Imports System.Windows
Imports System.Windows.Input
Imports System.Windows.Media

Namespace HitTesting
  Partial Public Class Window1
    Inherits System.Windows.Window
    Public Sub New()
      InitializeComponent()
    End Sub

    Private Sub OnMouseDown(ByVal sender As Object, _
      ByVal e As MouseEventArgs)
      ' Mausposition ermitteln
      Dim pt As Point = e.GetPosition(DirectCast(sender, UIElement))
      ' Fläche für Trefferbereich festlegen
      Dim newHitTestArea As EllipseGeometry = New EllipseGeometry(pt, 20.0, 20.0)
      ' Rückrufprozedur für einen Treffer einrichten
```

```
        VisualTreeHelper.HitTest(path, Nothing, _
                                New HitTestResultCallback(AddressOf HitResult), _
                                New GeometryHitTestParameters(newHitTestArea))
    End Sub

    Public Function HitResult(ByVal result As HitTestResult) As HitTestResultBehavior

        ' Ermitteln des Hit-Test-Ergebnisses
        Select Case DirectCast(result, GeometryHitTestResult).IntersectionDetail
            Case IntersectionDetail.FullyInside
                MessageBox.Show("Vollständig innen!")
                Return HitTestResultBehavior.Continue
            Case IntersectionDetail.FullyContains
                MessageBox.Show("Voll getroffen!")
                Return HitTestResultBehavior.Stop
            Case IntersectionDetail.Intersects
                MessageBox.Show("Teilweise getroffen!")
                Return HitTestResultBehavior.Continue
        End Select
        Return HitTestResultBehavior.Stop
    End Function
End Class
End Namespace
```

Listing 14.19 Das erweiterte Hit-Testing

Zusammenfassung

In diesem Kapitel haben Sie erfahren, wie vielfältig die verschiedenen Grafikelemente von Windows Presentation Foundation eingesetzt werden können. Entscheidend bei der Darstellung einer Benutzerschnittstelle ist die einfache Kombination von Steuerelementen und grafischen Elementen.

Viele interessante Vereinfachungen werden durch Anwendung der Path-Klasse ermöglicht. Die Definition und Zusammenfassung von grafischen Figuren wird mit einem Path-Element stark vereinfacht. Entscheidende Design-Möglichkeiten werden aber durch überall anwendbare Transformationen geschaffen. Es ist ohne großen Programmieraufwand mit WPF möglich, einerseits moderne Layouts für Spielprogramme zu definieren, aber andererseits auch eine vollständig in der Größe einstellbare Benutzerschnittstelle zu erstellen, die einen hohen Komfort beim Arbeiten mit dieser Applikation ermöglicht.

