15 Pfade, Bereiche und Clipping

Haben Sie sich schon einmal mit der Grafikprogrammierung in PostScript beschäftigt? Dann wissen Sie ja sicher, was ein *Grafikpfad* ist. In PostScript können Sie ohne Grafikpfade im Grunde gar nichts machen. In anderen grafischen Programmierumgebungen spielen die Pfade beim Zeichnen von Objekten eine nicht ganz so zentrale Rolle wie in PostScript, dennoch werden sie allgemein als wichtiges Hilfsmittel bei der Programmierung von Grafiken angesehen.

Ein Grafikpfad erfüllt eine im Grunde simple Aufgabe: Er bietet eine Möglichkeit, Geraden und Kurven miteinander zu verbinden. Wie Sie wisssen, können Sie mithilfe der *DrawLines*-Methode Geraden und mithilfe von *DrawBeziers* miteinander verbundene Bézier-Kurven zeichnen, bisher habe ich aber noch kein Verfahren vorgestellt, mit dem Sie Geraden und Bézier-Kurven miteinander verbinden können. Zu genau diesem Zweck dient der Pfad. Das klingt recht schlicht, eröffnet uns aber in Wirklichkeit Zugang zu einer Vielzahl von Zeichentechniken, die ich in diesem sowie in den Kapiteln 17 und 19 erläutern möchte.

Pfade können auch beim so genannten *Clipping* eingesetzt werden. Darunter versteht man die Begrenzung der Grafikausgabe auf einen bestimmten Bereich des Bildschirms oder der Druckseite. Wenn Sie einen Clippingpfad festlegen, wird dieser zunächst in einen *Bereich* (region) konvertiert. Ein Bereich beschreibt eine Fläche des Ausgabegeräts in Gerätekoordinaten.

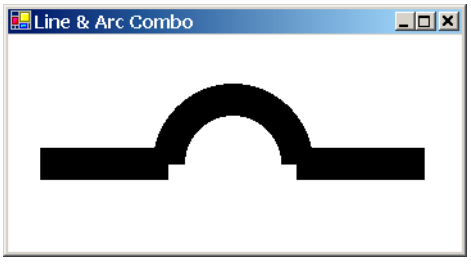
Ein Problem – eine Lösung

Beginnen wir die Untersuchung der Grafikpfade mit einem Problem der Grafikprogrammierung. Angenommen, Sie möchten eine Figur zeichnen, die aus einer Linie, einem Halbkreis und einer weiteren Linie besteht, die miteinander verbunden sind. Dabei möchten Sie einen Stift verwenden, der erheblich dicker als 1 Pixel ist. Versuchen wir es:

LineArcCombo.cs

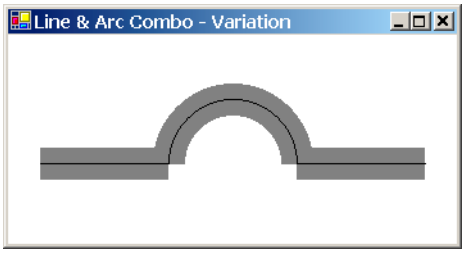
```
//-----------------------------------------
// LineArcCombo.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class LineArcCombo: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new LineArcCombo());
     }
     public LineArcCombo()
     {
          Text = "Line & Arc Combo";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Pen pen = new Pen(clr, 25);

          grfx.DrawLine(pen,  25, 100, 125, 100);
          grfx.DrawArc (pen, 125,  50, 100, 100, -180, 180);
          grfx.DrawLine(pen, 225, 100, 325, 100);
     }
}
```

Die beiden Linien sind 100 Einheiten lang (dies entspricht 100 Pixeln auf dem Bildschirm oder 1 Zoll auf dem Drucker); der Kreis, aus dem der Bogen gebildet wird, hat einen Durchmesser von 100 Einheiten. Der Stift ist 25 Einheiten breit. Und so sieht das Ergebnis aus:



Möglicherweise ist dies genau das Bild, das Sie brauchen. Es ist aber leider nicht das, was ich wollte. Ich wollte, dass Linien und Halbkreis verbunden sind. Zugegeben, die Linien und der Bogen berühren sich, aber sie sind definitiv nicht vernünftig verbunden. Am unteren Ende des Bogens befinden sich unerwünschte Aussparungen.

Ändern wir das Programm LineArcCombo so ab, dass die Figur zweimal gezeichnet wird: einmal mit einem dicken grauen und einmal mit einem schwarzen, 1 Pixel breiten Stift. So lässt sich leichter erkennen, was hier vor sich geht:



Die 25 Pixel breiten Linien ragen auf jeder Seite um 12 Pixel über die 1 Pixel breiten Linien hinaus. Da Linien und Bogen mithilfe separater Methodenaufrufe gezeichnet werden, ist jede Figur eine eigene Einheit. An den beiden Berührungspunkten überschneiden sich die breiten Striche zwar, bilden aber kein geschlossenes Ganzes.

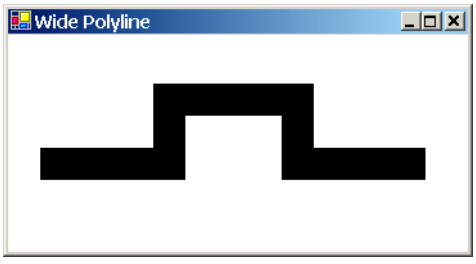
Sie könnten jetzt natürlich die Koordinaten so lange manipulieren, bis das Bild stimmt. Beispielsweise könnten Sie den Bogen um 12 Einheiten nach unten versetzen, oder so etwas in der Art. Aber tief im Inneren wissen Sie, dass Sie das Problem damit nicht gelöst, sondern nur verdrängt haben.

Wir benötigen eine Möglichkeit, dem Grafiksystem mitzuteilen, dass Bogen und Linien miteinander verbunden sein sollen. Wenn wir es mit Geraden zu tun hätten, wäre das ein Klacks: Statt getrennter Linien mit *DrawLine* könnten wir einfach mit *DrawLines* eine Polylinie zeichnen. Das folgende Programm zeichnet z.B. eine Figur, die dem gewünschten Ergebnis nahe kommt.

WidePolyline.cs

```
//-----------------------------------------
// WidePolyline.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class WidePolyline: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new WidePolyline());
     }
     public WidePolyline()
     {
          Text = "Wide Polyline";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          Pen pen = new Pen(clr, 25);
          grfx.DrawLines(pen, new Point[] {
                         new Point( 25, 100), new Point(125, 100),
                         new Point(125,  50), new Point(225,  50),
                         new Point(225, 100), new Point(325, 100) });
     }
}
```

Der *DrawLines*-Aufruf enthält ein Array aus sechs *Point*-Strukturen, um eine aus fünf Linien bestehene Polylinie anzuzeigen:



Das Grafiksystem weiß, dass diese Linien verbunden dargestellt werden sollen, da sie sich alle in einem Funktionsaufruf befinden. An den Berührungspunkten wird die breite Linie richtig gezeichnet.

Bei Verwendung einer Polylinie im Programm WidePolyline bietet sich für die Figur aus Geraden und Bogen eine andere Lösung an. Sie könnten beispielsweise, wie in Kapitel 5 beschrieben, eine Ellipse als Polylinie zeichen und den Bogen auf diese Weise implementieren. Oder Sie könnten sowohl die Geraden (indem Sie zwischen den Geradenendpunkten kollineare Kontrollpunkte angeben) als auch den Bogen in Bézier-Kurven umwandeln (mithilfe der Formeln aus Kapitel 13), und anschließend die ganze Figur mit *DrawBeziers* zeichnen.

Es muss aber einen direkteren Weg geben, dem Grafiksystem zu sagen, dass Geraden und Bogen miteinander verbunden sind. Wir brauchen so etwas wie *DrawLines*, eine Möglichkeit zur Kombination von Geraden und Bögen. Und wo wir schon dabei sind, können wir auch gleich noch verlangen, dass diese magische Funktion auch mit Bézier-Kurven bzw. allgemein mit Kurven umgehen kann.

Diese magische Funktion (genauer gesagt handelt es sich um eine magische Klasse) heißt *GraphicsPath*. Das nächste Programm zeichnet die Figur wie gewünscht und benötigt dafür nur drei Anweisungen mehr als LineArcCombo.

LineArcPath.cs

```
//-----------------------------------------
// LineArcPath.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
class LineArcPath: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new LineArcPath());
     }
     public LineArcPath()
     {
          Text = "Line & Arc in Path";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          GraphicsPath path = new GraphicsPath();
          Pen          pen  = new Pen(clr, 25);
```

```
        path.AddLine( 25, 100, 125, 100);
        path.AddArc (125,  50, 100, 100, -180, 180);
        path.AddLine(225, 100, 325, 100);

        grfx.DrawPath(pen, path);
    }
}
```

Eine der drei zusätzlichen Anweisungen erstellt zu Beginn der *DoPage*-Methode den Pfad:

```
GraphicsPath path = new GraphicsPath();
```

Die Klasse zur Implementierung des Pfads heißt zwar *GraphicsPath,* ich werde aber für die Instanzen dieser Klasse den einfacheren Variablennamen *path* verwenden. *GraphicsPath* ist im Namespace *System.Drawing.Drawing2D* definiert; eine weitere *using*-Anweisung ermöglicht die Verwendung der drei zusätzlichen Anweisungen in diesem Programm.

Das Programm LineArcCombo zeichnete die erste Linie mithilfe der *DrawLine*-Methode der *Graphics*-Klasse:

```
grfx.DrawLine(pen, 25, 100, 125, 100);
```

Das Programm LineArcPath ersetzt diese Anweisung durch die *AddLine*-Methode der *GraphicsPath*-Klasse:

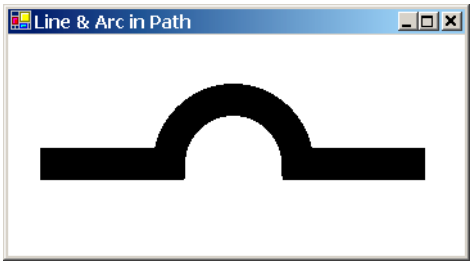
```
path.AddLine(25, 100, 125, 100);
```

Abgesehen von dem *Pen*-Argument, das in der *AddLine*-Methode nicht enthalten ist, unterscheiden sich die Argumente nicht von denen der *DrawLine*-Methode. Das Gleiche gilt für *AddArc* im Vergleich zu *DrawArc.* Die Aufrufe *AddLine* und *AddArc* führen keine Zeichenoperation aus, sondern liefern nur Koordinaten, die im Pfad gesammelt werden.

Zum Schluss wird der Pfad mit der dritten zusätzlichen Anweisung auf dem Bildschirm ausgegeben:

```
grfx.DrawPath(pen, path);
```

DrawPath ist übrigens eine Methode unserer guten alten Freundin, der *Graphics*-Klasse. Der *DrawPath*-Aufruf liefert genau das Bild, das wir uns vorgestellt haben:



Erfahrenen Win32-API- oder MFC-Programmierern wird aufgefallen sein, dass die Implementierung eines Grafikpfads in Windows Forms einem anderen Konzept folgt. In der Win32-API versetzt die *BeginPath*-Funktion den Gerätekontext in einen besonderen Modus, in dem Aufrufe der üblichen Zeichenfunktionen (beispielsweise *LineTo, BezierTo* u.ä.) nicht ausgegeben, sondern in den Pfad eingefügt werden. Der Pfad wird mit einem Aufruf von *EndPath* abgeschlossen und anschließend mit einem Aufruf von *StrokePath* gezeichnet (oder anderweitig verwendet).

Der Ansatz in Windows Forms ist wesentlich flexibler. In der Win32-API gibt es für einen bestimmten Gerätekontext immer nur einen Pfad, Windows Forms dagegen ermöglicht das Erstellen und Speichern beliebig vieler Pfade. Darüber hinaus ist zum Erstellen eines Pfads kein

Graphics-Objekt erforderlich. Der Pfad existiert unabhängig von einem *Graphics*-Objekts, bis er mithilfe von *DrawPath* angezeigt (oder anderweitig weiterverarbeitet) wird.

Sie könnten das *Graphics*-Objekt im Programm LineArcPath auch als Feld speichern. Dann könnten Sie die Erstellung des Pfads und die Aufrufe von *AddLine* und *AddArc* im Formularkonstruktor ausführen. Dadurch würde die *DoPage*-Methode nur noch den Stift erstellen und *DrawPath* aufrufen. Wenn Sie den Formularkonstruktor so richtig freiräumen möchten, können Sie die Anweisung zur Stifterstellung auch als Formularfeld machen.

Der Pfad in aller Form

Zu Beginn dieses Abschnitts möchte ich einige Begriffe definieren und einen kurzen Überblick über das Thema Pfade geben:

Unter einem *Pfad* versteht man eine Sammlung geräteunabhängiger Koordinatenpunkte zur Beschreibung von Geraden und Kurven. Diese Geraden und Kurven können, müssen aber nicht miteinander verbunden sein. Ein Satz verbundener Linien und Kurven innerhalb des Pfads wird als *Teilpfad* oder *Figur* bezeichnet. (Beide Begriffe werden in der Windows Forms-Schnittstelle synonym verwendet.) Ein Pfad besteht also aus null oder mehr Teilpfaden. Jeder Teilpfad ist eine Sammlung aus verbundenen Linien und Kurven. Der im Programm LineArcPath erstellte Pfad verfügt nur über einen einzigen Teilpfad.

Ein Teilpfad kann entweder *offen* oder *geschlossen* sein. Ein Teilpfad ist geschlossen, wenn der Endpunkt der letzten Linie des Teilpfads mit dem Anfangspunkt der ersten Linie verbunden ist. (Zum Schließen eines Teilpfads steht in der *GraphicsPath*-Klasse die spezielle Methode *CloseFigure* zur Verfügung.) Andernfalls handelt es sich um einen offenen Teilpfad. Der Teilpfad in LineArcPath ist offen.

Ich habe Ihnen die *DrawPath*-Methode bereits vorgestellt, mit der die Linien und Kurven gezeichnet werden, aus denen sich der Pfad auf einem Ausgabegerät zusammensetzt. Die *Graphics*-Klasse enthält auch eine *FillPath*-Methode, die mit einem Pinsel die Innenbereiche aller geschlossenen Teilpfade füllt. Für die *FillPath*-Methode gelten *alle* offenen Teilpfade als geschlossen und definieren somit geschlossene Flächen.

Ein Pfad kann auch in einen *Bereich* konvertiert werden. Dies werde ich im Verlauf dieses Kapitels noch zeigen. Im Gegensatz zu einen Pfad (einer Sammlung aus Linien und Kurven) beschreibt ein Bereich einen Ausschnitt der Bildschirmoberfläche. Dieser Bereich kann ein einfaches Rechteck oder ein sehr komplexes Gebilde sein. Die durch den Bereich definierte Fläche kann mit einem Pinsel gefüllt oder als Clippingbereich eingesetzt werden. Durch das so genannte Clipping wird der Zeichenvorgang auf einen bestimmten Bereich der Bildschirmoberfläche begrenzt.

Einige Programmierer erliegen bei den ersten Gehversuchen mit Grafikpfaden dem Irrglauben, dabei müsse es sich um viel mehr handeln als eine bloße Ansammlung aus Linien- und Kurvendefinitionen. Wir wollen uns mit einem Blick auf die *GraphicsPath*-Eigenschaft ein für alle Mal von diesem Gedanken verabschieden. Ein Pfad enthält keinerlei dauerhafte Daten, auf die nicht über die Pfadeigenschaften zugegriffen werden könnte:

GraphicsPath-Eigenschaften

Typ	Eigenschaft	Zugriff	Beschreibung
FillMode	*FillMode*	get/set	*FillMode.Alternate* oder *FillMode.Winding*
int	*PointCount*	get	Anzahl der Punkte im Pfad
PointF[]	*PathPoints*	get	Array aus Koordinatenpunkten
byte[]	*PathTypes*	get	Array aus Punkttypen
PathData	*PathData*	get	Dupliziert *PathPoints* und *PathTypes*

Die Enumeration *FillMode* wird auch mit der in Kapitel 5 beschriebenen *DrawPolygon*-Methode und der *DrawClosedCurve*-Methode (siehe Kapitel 13) verwendet. Bei Pfaden gibt die *FillMode*-Eigenschaft an, auf welche Weise der Pfad gefüllt (oder in einen Bereich konvertiert) wird, wenn er sich überschneidende Linien enthält.

Die anderen vier Eigenschaften sind redundant und definieren nichts weiter als zwei Arrays von gleicher Größe:

- Ein Array aus *PointF*-Strukturen namens *PathPoints*
- Ein Array aus *byte*-Werten namens *PathTypes*

Die Anzahl der Elemente in diesen Array (die aus *PathPoints.Length* oder *PathTypes.Length* abgerufen werden können) steht auch in der *PointCount*-Eigenschaft zur Verfügung.

Die *PathData*-Eigenschaft sorgt ebenfalls für Redundanz. Bei dieser Eigenschaft handelt es sich um ein Objekt vom Typ *PathData,* das im Namespace *System.Drawing.Drawing2D* definiert ist. Die Klasse *PathData* besitzt folgende beiden Eigenschaften:

PathData-Eigenschaften

Typ	Eigenschaft	Zugriff	Beschreibung
PointF[]	*Points*	get/set	Array aus Koordinatenpunkten
byte[]	*Types*	get/set	Array aus Punkttypen

Das *Points*-Array der *PathData*-Eigenschaft ist für alle *GraphicsPath*-Objekte identisch mit der *PathPoints*-Eigenschaft; das *Types*-Array der *PathData*-Eigenschaft stimmt mit der *PathTypes*-Eigenschaft überein.

Bei den *byte*-Werten, aus denen die *PathTypes*-Eigenschaft besteht, handelt es sich um Werte der Enumeration *PathPointType,* die ebenfalls in *System.Drawing.Drawing2D* definiert ist:

PathPointType-Enumeration

Member	Wert
Start	0
Line	1
Bezier oder *Bezier3*	3
PathTypeMask	7
DashMode	16
PathMarker	32
CloseSubpath	128

Jede *PointF*-Struktur im *PathPoints*-Array ist mit einem der *PathPointType*-Werte *Start, Line* oder *Bezier* verknüpft. Der *Start*-Typ gibt den ersten Punkt einer Figur an, ein mit dem *Line*-Typ angegebener Punkt eine Gerade, der *Bezier*-Typ gibt einen zu einer Bézier-Kurve gehörenden Punkt an. Wenn Sie zum Pfad Bögen oder kanonische Spline-Kurven hinzufügen, werden diese in Bézier-Kurven umgewandelt. Diese Art der Umwandlung sollte Ihnen einleuchtend vorkommen, nachdem ich Ihnen in Kapitel 13 demonstriert habe, dass Kreise sehr genau mit Bézier-Kurven gezeichnet werden können.

Bei den letzten drei Werten der *PathPointType*-Enumeration handelt es sich um Flags, die mit den Werten von *Start, Line* oder *Bezier* kombiniert werden können. Wie wir noch feststellen werden, werden die *PathMarker*- und *CloseSubpath*-Flags durch Aufrufe der *GraphicsPath*-Methode erzeugt.

Das Enumerationsmember *PathTypeMask* stellt eine Bitmaske dar, mit deren Hilfe die Werte nach Punkttypen (*Start, Line* oder *Bezier)* und Flags (*DashMode, PathMarker* oder *CloseSubpath)* unterschieden werden können.

Ein Pfad stellt allerdings keine Möglichkeit zur Verfügung, diese Koordinatenpunkte mit Maßeinheiten der realen Welt in Verbindung zu bringen. Es erübrigt sich damit die Frage, ob es sich bei den Koordinatenpunkten eines Pfads um Pixel-, Zoll-, Millimeter- oder sonstige Werte handelt. Es sind ganz einfach Punkte. Sie werden erst dann in Pixel, Zoll oder Millimeter umgewandelt, wenn der Pfad auf einem Ausgabegerät dargestellt wird.

Einen Pfad erstellen

Die Klasse *GraphicsPath* verfügt über sechs Konstruktoren:

GraphicsPath-Konstruktoren

```
GraphicsPath()
GraphicsPath(Point[] apt, byte[] abyPointType)
GraphicsPath(PointF[] aptf, byte[] abyPointType)
GraphicsPath(FillMode fm)
GraphicsPath(Point[] apt, byte[] abyPointType, FillMode fm)
GraphicsPath(PointF[] aptf, byte[] abyPointType, FillMode fm)
```

Wenn das *FillMode*-Argument nicht angegeben wird, lautet die Standardmethode *FillMode.Alternate.*

Vier dieser Konstruktoren legen die durchaus richtige Vermutung nahe, dass ein Pfad mithilfe eines Arrays aus *Point*- oder *PointF*-Strukturen und eines Arrays aus entsprechenden *PathPointType*-Enumerationswerten (die als Array aus *byte*-Werten ausgedrückt werden) erstellt werden kann. Es ist allerdings eher unwahrscheinlich, dass ein Programm einen neuen Pfad auf diese Weise erstellt. Diese Konstruktoren eignen sich viel besser dazu, die *PathPoints*-Werte bereits vorhandener Pfade zu verändern.

Ein neuer Pfad wird meistens über den Standardkonstruktor erstellt:

```
GraphicsPath path = new GraphicsPath();
```

Anschließend werden die Methoden der Klasse *GraphicsPath* aufgerufen, die Geraden und Kurven zum Pfad hinzufügen. Diese Methoden gleichen den entsprechenden Methoden der *Graphics*-Klasse, beginnen allerdings nicht mit dem Wort *Draw,* sondern mit *Add* und besitzen kein *Pen*-Argument.

Die folgenden *GraphicsPath*-Methoden fügen Geraden, Bézier-Kurven, Ellipsen und kanonische Spline-Kurven zum aktuellen Teilpfad hinzu. Ich habe in der Tabelle die Argumente weggelassen, da sie zum größten Teil mit den entsprechenden *Draw*-Methoden der *Graphics*-Klasse übereinstimmen:

GraphicsPath-Methoden (Auswahl)

```
void AddLine(...)
void AddLines(...)
void AddArc(...)
void AddBezier(...)
void AddBeziers(...)
void AddCurve(...)
```

Bögen und kanonische Spline-Kurven werden beim Einfügen in einen Pfad in Bézier-Kurven umgewandelt.

Wenn es sich bei *path* um ein Objekt vom Typ *GraphicsPath* handelt, fügen die folgenden drei Aufrufe drei verbundene Linien zum Pfad hinzu:

```
path.AddLine(0, 0, 0, 100);
path.AddLine(0, 100, 100, 100);
path.AddLine(100, 100, 100, 0);
```

Die Linien ergeben die rechte, untere und linke Seite eines Quadrats. Ich habe die Koordinaten so gewählt, dass der Endpunkt einer Linie jeweils mit dem Startpunkt der nächsten Linie übereinstimmt – gerade so, als würde ich sie zeichnen, ohne den Stift abzusetzen.

Bei der Definition eines Pfads müssen Sie allerdings nicht ganz so akribisch vorgehen. Sofern nicht anders angegeben (worauf ich gleich noch zurückkomme), gehören alle zum Pfad hinzugefügten Linien, Bögen, Bézier-Kurven und kanonischen Splines zu ein und derselben Figur. Falls die Koordinaten nicht exakt übereinstimmen sollten, verbindet der Pfad die Einzelteile automatisch mit einer Geraden. Sie können das gleiche Ergebnis erzielen wie mit den soeben gezeigten drei Anweisungen, indem Sie die zweite Anweisung einfach weglassen:

```
path.AddLine(0, 0, 0, 100);
path.AddLine(100, 100, 100, 0);
```

Da die erste Linie bei (0, 100) endet und die zweite bei (100, 100) beginnt, fügt der Pfad automatisch eine Gerade zwischen diesen beiden Punkten ein.

Sie können auch die folgenden drei Methoden aufrufen:

GraphicsPath-Methoden (Auswahl)

```
void StartFigure()
void CloseFigure()
void CloseAllFigures()
```

Diese drei Methodenaufrufe beenden den aktuellen Teilpfad und beginnen einen neuen. Darüber hinaus schließt *CloseFigure* den aktuellen Teilpfad. Dabei wird dem Pfad gegebenenfalls automatisch eine Gerade vom letzten bis zum ersten Punkt des Teilpfads hinzugefügt. *CloseAllFigures* schließt alle bislang zum Pfad gehörigen Teilpfade.

Die Aufrufe

```
path.AddLine(0, 0, 0, 100);
path.AddLine(0, 100, 100, 100);
path.AddLine(100, 100, 100, 0);
path.AddLine(100, 0, 0, 0);
path.CloseFigure();
```

erstellen explizit eine geschlossene quadratische Figur. Die Aufrufe

```
path.AddLine(0, 0, 0, 100);
path.AddLine(100, 100, 100, 0);
path.CloseFigure();
```

erzwingen, dass der Pfad für die obere und untere Seite automatisch eine Linie einfügt und erzeugen so die gleiche geschlossene Figur. Die Aufrufe

```
path.AddLine(0, 0, 0, 100);
path.AddLine(0, 100, 100, 100);
path.AddLine(100, 100, 100, 0);
path.AddLine(100, 0, 0, 0);
path.StartFigure();
```

erstellen eine Figur, die aus den vier Seiten eines Quadrats besteht, aber nicht geschlossen ist, da der Aufruf von *CloseFigure* fehlt.

Die folgenden Methoden beginnen eine neue Figur, die dann geschlossen wird:

GraphicsPath-Methoden (Auswahl)

```
void AddRectangle(...)
void AddRectangles(...)
void AddPolygon(...)
void AddEllipse(...)
void AddPie(...)
void AddClosedCurve(...)
```

Diese Aufrufe beispielsweise:

```
path.AddLine(0, 0, 100, 0);
path.AddRectangle(new Rectangle(50, 50, 100, 100);
path.AddLine(200, 0, 0, 0);
```

erstellen drei Teilpfade:

- Eine nicht geschlossene Linie
- Vier geschlossene Linien
- Eine nicht geschlossene Linie

Pfade können auch zu anderen Pfaden hinzugefügt werden:

GraphicsPath AddPath-Methoden

```
void AddPath(GraphicsPath path, bool bConnect)
```

Das zweite Argument gibt an, ob der hinzugefügte Pfad mit dem aktuellen Teilpfad verbunden werden soll.

Die *AddString*-Methoden fügen eine Textzeichenfolge zum Pfad hinzu. Die Syntax dieser Methoden unterscheidet sich wesentlich von der Syntax der *DrawString*-Methoden:

***GraphicsPath AddString*-Methoden**

```
void AddString(string str, FontFamily ff, int iStyle, float fSize,
               Point pt, StringFormat sf)
void AddString(string str, FontFamily ff, int iStyle, float fSize,
               PointF ptf, StringFormat sf)
void AddString(string str, FontFamily ff, int iStyle, float fSize,
               Rectangle rect, StringFormat sf)
void AddString(string str, FontFamily ff, int iStyle, float fSize,
               RectangleF rectf, StringFormat sf)
```

Die hier vorhandenen Argumente sehen nicht im Geringsten aus wie Koordinatenpunkte, dennoch fügen auch diese Methoden nichts anderes zum Pfad hinzu als eine Reihe von Geraden und Bézier-Kurven. Bei diesen Linien und Kurven handelt es sich um die Umrisse der Schriftzeichen.

Die Argumente von *AddString* sind gar nicht so eigenartig wie die Methodendefinitionen vermuten lassen. Das dritte Argument ist zwar als *int*-Wert definiert, es handelt sich aber in Wirklichkeit um ein Member der Enumeration *FontStyle* (*Regular, Bold, Italic, Underline* oder *Strikeout).* Daher entsprechen das zweite, dritte und vierte Argument den drei Argumenten im *Font*-Konstruktor.

Warum aber setzen die *AddString*-Methoden *Font*-Argumente nicht genauso ein wie *DrawString*? Das liegt daran, dass ein *Font* meist über eine bestimmte Punktgröße verfügt und ein Pfad keine Angaben zu den Abmessungen beinhaltet. Bei dem *AddString*-Argument *fSize* handelt es sich nicht um eine Punktgröße. Durch Festlegen des *fSize*-Arguments in *AddString* können Sie das Gleiche erreichen wie durch das Erstellen eines *Font*-Objekts mit einer bestimmten Pixelgröße und dem Argument *GraphicsUnit.Pixel* oder *GraphicsUnit.World* (wie in Kapitel 9 beschrieben). Der Text verfügt erst bei der Ausgabe über eine messbare Größe.

Durch Einfügen von Text in einen Pfad lassen sich so viele unglaubliche Effekte erzielen, dass ich das Kapitel 19, das sich damit beschäftigt, »Schriftspielereien« genannt habe.

Sie können in einen Pfad auch Markierungen einfügen, die nicht gezeichnet werden:

***GraphicsPath*-Methoden (Auswahl)**

```
void SetMarkers()
void ClearMarkers()
```

Mithilfe der Klasse *GraphicsPathIterator* können Sie nach diesen Markierungen suchen, wodurch das Bearbeiten eines Pfads erleichtert wird.

Einen Pfad darstellen

Die Ausgabe eines Pfads erfolgt meist durch Aufruf einer der beiden folgenden Methoden der *Graphics*-Klasse:

***Graphics*-Methoden (Auswahl)**

```
void DrawPath(Pen pen, GraphicsPath path)
void FillPath(Brush brush, GraphicsPath path)
```

Die *DrawPath*-Methode zeichnet unter Verwendung des angegebenen Stifts die Linien und Kurven, aus denen sich der Pfad zusammensetzt. *FillPath* füllt die Innenflächen aller geschlossenen

Teilpfade mit dem festgelegten Pinsel. Zu diesem Zweck schließt die Methode vorübergehend alle nicht geschlossenen Teilpfade; diese Änderungen wirken sich allerdings nicht dauerhaft auf den Pfad aus. Sollten sich Linien des Pfads überschneiden, erfolgt die Füllung der Innenflächen auf Basis der aktuellen *FillPath*-Eigenschaft des *GraphicsPath*-Objekts. Zum Zeitpunkt der Ausgabe werden die Pfadpunkte allen Transformationen unterworfen, die für das *Graphics*-Objekt aktiviert wurden.

Wir wollen uns das Ganze einmal in der Praxis anschauen. Das Programm Flower zeichnet unter Verwendung eines Pfads und einer Transformation eine Blume.

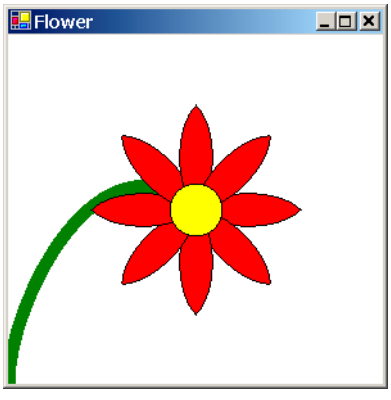
Flower.cs

```
//-----------------------------------
// Flower.cs © 2001 by Charles Petzold
//-----------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class Flower: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new Flower());
     }
     public Flower()
     {
          Text = "Flower";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
               // Grünen Stengel von der linken unteren Ecke zur Mitte zeichnen

          grfx.DrawBezier(new Pen(Color.Green, 10),
                    new Point(0, cy),          new Point(0, 3 * cy / 4),
                    new Point(cx / 4, cy / 4), new Point(cx / 2, cy / 2));

               // Transformationen für den Rest der Blume festlegen

          float fScale = Math.Min(cx, cy) / 2000f;
          grfx.TranslateTransform(cx / 2, cy / 2);
          grfx.ScaleTransform(fScale, fScale);

               // Rote Blütenblätter zeichnen

          GraphicsPath path = new GraphicsPath();

          path.AddBezier(new Point(  0,    0), new Point(125,  125),
                         new Point(475,  125), new Point(600,    0));
          path.AddBezier(new Point(600,    0), new Point(475, -125),
                         new Point(125, -125), new Point(  0,    0));

          for (int i = 0; i < 8; i++)
          {
               grfx.FillPath(Brushes.Red, path);
               grfx.DrawPath(Pens.Black, path);
               grfx.RotateTransform(360 / 8);
          }
```

```
            // Gelben Kreis in der Mitte zeichnen

        Rectangle rect = new Rectangle(-150, -150, 300, 300);
        grfx.FillEllipse(Brushes.Yellow, rect);
        grfx.DrawEllipse(Pens.Black, rect);
    }
}
```

Die *DoPage*-Methode zeichnet zunächst von der linken unteren Ecke bis zur Mitte des Clientbereichs (oder der Druckerseite) eine Bézier-Kurve. Diese bildet den Stengel. Danach wird eine Welttransformation eingerichtet, die eine aus vier Quadranten bestehenden isotrope Zeichenfläche definiert, bei welcher der Ursprung im Mittelpunkt liegt und die Koordinaten von –1000 bis 1000 reichen.

Nun muss das Programm die Blütenblätter zeichnen, und an diesem Punkt kommt der Pfad ins Spiel. Wenn die Blätter eine elliptische Form hätten, könnten wir hier einfach *FillEllipse* verwenden. Da die Form der Blütenblätter aber nicht ganz einer Ellipse entspricht, lassen sie sich exakter durch zwei Bézier-Kurven definieren. Das Füllen einer solchen Figur erfordert einen Pfad. Nach der Erstellung des Pfads ruft das Programm achtmal die Methoden *FillPath* und *DrawPath* auf. Nach jedem Aufruf dieser beiden Methoden ändert der Aufruf von *RotateTransform* die Welttransformation des *Graphics*-Objekts, sodass die acht Blütenblätter kreisförmig um die Mitte angeordnet werden. Als Letztes zeichnet *DoPage* in der Mitte des Clientbereichs einen gelben Kreis.



Sie erinnern sich sicher noch an das Programm Scribble aus Kapitel 8. Dort habe ich Ihnen gezeigt, wie vom Benutzer gezeichnete Linien mithilfe der Klasse *ArrayList* gespeichert werden. Bei dieser Klasse handelt es sich um ein arrayähnliches Objekt, das seine Größe dynamisch anpassen kann. Der Einsatz der *ArrayList*-Klasse erfolgt auf fast die gleiche Weise wie das Speichern von Koordinaten in einem Pfad. Die Verwendung eines *GraphicsPath*-Objekts anstelle eines *ArrayList*-Objekts führt zu einer erheblichen Vereinfachung des Programms. Dies Programm ist sogar noch einfacher als die ScribbleWithBitmap-Version aus Kapitel 11, die das Bild in einer Schattenbitmap speichert.

ScribbleWithPath.cs

```
//---------------------------------------------
// ScribbleWithPath.cs © 2001 by Charles Petzold
//---------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ScribbleWithPath: Form
{
     GraphicsPath path;
     bool         bTracking;
     Point        ptLast;

     public static void Main()
     {
          Application.Run(new ScribbleWithPath());
     }
     public ScribbleWithPath()
     {
          Text = "Scribble with Path";
          BackColor = SystemColors.Window;
          ForeColor = SystemColors.WindowText;

               // Pfad erstellen

          path = new GraphicsPath();
     }
     protected override void OnMouseDown(MouseEventArgs mea)
     {
          if (mea.Button != MouseButtons.Left)
               return;

          ptLast = new Point(mea.X, mea.Y);
          bTracking = true;

               // Beginn einer Figur

          path.StartFigure();
     }
     protected override void OnMouseMove(MouseEventArgs mea)
     {
          if (!bTracking)
               return;

          Point ptNew = new Point(mea.X, mea.Y);

          Graphics grfx = CreateGraphics();
          grfx.DrawLine(new Pen(ForeColor), ptLast, ptNew);
          grfx.Dispose();

              // Linie hinzufügen

          path.AddLine(ptLast, ptNew);

          ptLast = ptNew;
     }
```

```
    protected override void OnMouseUp(MouseEventArgs mea)
    {
        bTracking = false;
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
            // Pfad zeichnen

        pea.Graphics.DrawPath(new Pen(ForeColor), path);
    }
}
```

Neben einer zusätzlichen *using*-Anweisung habe ich in dieser ScribbleWithPath-Version des Scribble-Programms (das nicht über eine Speichermöglichkeit verfügt) einen Pfad als Feldvariable gespeichert und anschließend einfach vier Anweisungen hinzugefügt (diese sind durch Kommentare gekennzeichnet).

Die Pfaderstellung erfolgt im Konstruktor. Wenn der Mauscursor auf dem Formularclientbereich positioniert ist, führt jedes Drücken der linken Maustaste zu einem Aufruf der *StartFigure*-Methode, die einen neuen Pfad beginnt. Ein *AddLine*-Aufruf während der *OnMouseMove*-Methode fügt eine neue Linie zum Pfad hinzu. Die *OnPaint*-Methode besteht hier nur noch aus einem *DrawPath*-Aufruf.

Pfadtransformationen

Die Klasse *GraphicsPath* enthält mehrere Methoden, mit denen das Programm Pfade verändern kann. Die erste dieser Methoden wirkt zunächst vermutlich ein bisschen verwirrend. (Zumindest ging es mir so, als sie mir das erste Mal über den Weg lief.)

Transform-Methode der *GraphicsPath*-Klasse

```
void Transform(Matrix matrix)
```

Wie Sie aus Kapitel 7 wissen, besitzt die *Graphics*-Klasse eine Eigenschaft vom Typ *Matrix,* die den Namen *Transform* trägt. Diese Eigenschaft wirkt sich auf jede weitere Grafikausgabe aus.

Mit *Transform* in *GraphicsPath* verhält es sich anders. Hierbei handelt es sich nicht um eine Eigenschaft, sondern um eine Methode. Und dieser Unterschied ist von großer Bedeutung. Eine Eigenschaft ist üblicherweise ein Merkmal eines Objekts, eine Methode dagegen führt eine Operation durch. Man könnte eine Eigenschaft mit einem Adjektiv vergleichen; eine Methode wäre dementsprechend ein Verb.

Die *Transform*-Methode der *GraphicsPath*-Klasse ändert die Koordinaten des Pfads dauerhaft, indem sie die angegebene Transformation auf diese Koordinaten anwendet. Sie wirkt sich nicht auf Koordinaten aus, die erst später zum Pfad hinzugefügt werden. Darüber hinaus behält das *GraphicsPath*-Objekt die Transformationen nicht bei. Wenn Sie beispielsweise über ein *Matrix*-Objekt namens *matrix* verfügen, mit dem der Wert eines Koordinatenpunkts verdoppelt wird und Sie folgenden Aufruf ausführen,

```
grfx.Transform(matrix);
```

erhalten Sie dasselbe Ergebnis, als würden Sie mithilfe der *PathPoints*-Eigenschaft das Array aus den Pfadkoordinaten abrufen, alle Arraywerte verdoppeln und auf der Grundlage dieser veränderten Punkte einen neuen Pfad erstellen.

Die *Transform*-Methode ist die einzige in der *GraphicsPath*-Klasse, die Matrizentransformationen verarbeiten kann. Zu diesem Zweck benötigen Sie die *Matrix*-Klasse, die im Namespace *System.Drawing.Drawing2D* definiert ist. (Ich bin gegen Ende von Kapitel 7 bereits kurz auf diese Klasse eingegangen.) Am einfachsten setzen Sie die *Matrix*-Klasse ein, indem Sie zuerst mithilfe des Standardkonstruktors eine Identitätsmatrix erstellen:

```
Matrix matrix = new Matrix();
```

Anschließend stehen Ihnen verschiedene Methoden der *Matrix*-Klasse zur Änderung dieser Transformation zur Verfügung. Die *Translate*-Methode entspricht der *TranslateTransform*-Methode der *Graphics*-Klasse. (Ich wäre ehrlich gesagt nicht sonderlich überrascht, wenn die *Graphics*-Klasse ihre *TranslateTransform*-Methoden durch einen einfachen Aufruf der entsprechenden *Translate*-Methode ihrer *Transform*-Eigenschaft implementierte.)

Translate-Methoden der *Matrix*-Klasse

```
void Translate (float dx, float dy)
void Translate (float dx, float dy, MatrixOrder mo)
```

Die Enumeration *MatrixOrder* verfügt über zwei Member: *Append* und *Prepend*.

Die *Scale*-Methode führt zum gleichen Ergebnis wie die *ScaleTransform*-Methode der *Graphics*-Klasse.

Scale-Methoden der *Matrix*-Klasse

```
void Scale(float sx, float sy)
void Scale(float sx, float sy, MatrixOrder mo)
```

Vor kurzem habe ich das Verdoppeln von Pfadkoordinaten erwähnt. Dieser Vorgang lässt sich mit folgenden Codezeilen durchführen:

```
Matrix matrix = new Matrix();
matrix.Scale(2, 2);
path.Transform(matrix);
```

Die *Matrix*-Klasse enthält darüber hinaus auch eine *Rotate*-Methode:

Rotate-Methoden der *Matrix*-Klasse

```
void Rotate(float fAngle)
void Rotate(float fAngle, MatrixOrder mo)
```

Sie können das Programm Flower so abändern, das nicht die *RotateTransform*-Methode der *Graphics*-Klasse, sondern die *Rotate*-Methode der Klasse *Matrix* verwendet wird. Erstellen Sie dazu nach dem Pfad ein *Matrix*-Objekt, das eine Rotation um 45 Grad beschreibt:

```
Matrix matrix = new Matrix();
matrix.Rotate(45);
```

Anschließend rufen Sie in der *for*-Schleife nicht die *RotateTransform*-, sondern die *Transform*-Methode des Pfads auf:

```
path.Transform(matrix);
```

In der ursprünglichen Flower-Version bleibt der Pfad erhalten und der *RotateTransform*-Aufruf bestimmt die Transformation der Koordinaten während der Pfadausgabe durch die *Graphics*-Klasse. In der jetzigen Version werden die im Pfad gespeicherten Koordinaten rotiert. Am Ende

der *for*-Schleife sind acht Rotationsvorgänge um jeweils 45 Grad erfolgt und die Pfadkoordinaten zu ihren Ursprungswerten zurückgekehrt.

Für folgende interessante Methode der *Matrix*-Klasse gibt es in der *Graphics*-Klasse keine Entsprechung:

RotateAt-Methoden der *Matrix*-Klasse

```
void RotateAt(float fAngle, PointF ptf)
void RotateAt(float fAngle, PointF ptf, MatrixOrder mo)
```

Durch eine Matrixtransformation wird ein Bild normalerweise um den Punkt (0, 0) rotiert. Diese Methode ermöglicht Ihnen jedoch die Angabe eines Punkts, der als Rotationszentrum verwendet werden soll. Angenommen, Sie erstellen folgenden Pfad:

```
GraphicsPath path = new GraphicsPath();
path.AddRectangle(new Rectangle(0, 0, 100, 100));
```

Der Pfad enthält die Punkte (0, 0), (100, 0), (100, 100) und (0, 100). Erstellen Sie ein *Matrix*-Objekt, rufen Sie die *Rotate*-Methode mit einem Rotationswinkel von 45 Grad auf und wenden Sie sie auf den Pfad an:

```
Matrix matrix = new Matrix();
matrix.Rotate(45);
path.Transform(matrix);
```

Die Punkte in diesem Pfad lauten gerundet (0, 0), (70.7, 70.7), (0, 141.4) und (–70, 70). Wenn Sie stattdessen für die *RotateAt*-Methode den Mittelpunkt des Rechtecks angeben:

```
Matrix matrix = new Matrix();
matrix.RotateAt(45, new Point(50, 50));
path.Transform(matrix);
```

enthält der Pfad die Punkte (50, –20.7), (120.7, 50), (50, 120.7) und (–20.7, 50).

Die *Matrix*-Klasse enthält auch eine Methode zur Scherung:

Shear-Methoden der *Matrix*-Klasse

```
void Shear(float xShear, float yShear)
void Shear(float xShear, float yShear, MatrixOrder mo)
```

Wenn diese Methoden auf eine Standardtransformation angewendet werden, ergeben sich diese Transformationsformeln:

$x' = x + xShear \cdot y$
$y' = yShear \cdot x + y$

Weitere Veränderungen des Pfads

Transform ist nicht die einzige Methode der Klasse *GraphicsPath,* mit der sämtliche Koordinaten eines Pfads verändert werden können. Die *Flatten*-Methode beispielsweise konvertiert alle Bézier-Kurven eines Pfads in gerade Liniensegmente:

Flatten-Methoden der *GraphicsPath*-Klasse

```
void Flatten()
void Flatten(Matrix matrix)
void Flatten(Matrix matrix, float fFlatness)
```

Optional kann auf die Punkte zuerst eine *Matrix*-Transformation angewendet werden.

Je höher der Wert des Arguments *fFlatness,* desto weniger Liniensegmente enthält der Pfad. Das Argument *fFlatness* ist standardmäßig mit einem Wert von 0,25 definiert. Der Wert 0 kann nicht definiert werden.

Die *Widen*-Methode hat durchgreifendere Änderungen des Pfads zur Folge als *Flatten.* Beim ersten Argument dieser Methode handelt es sich stets um ein *Pen*-Objekt:

Widen-Methoden der *GraphicsPath*-Klasse

```
void Widen(Pen pen)
void Widen(Pen pen, Matrix matrix)
void Widen(Pen pen, Matrix matrix, float fFlatness)
```

Die Methode ignoriert die Farbe des Stifts und verwendet nur die Stiftbreite, die zumeist einen Mindestwert von einigen Einheiten aufweist. Stellen wir uns Folgendes vor: Ein Pfad wird mit einem breiten Stift gezeichnet. Der neue Pfad ist der Umriss dieser breiten Linie. Jeder offene Pfad wird geschlossen, jeder geschlossene Pfad wird in zwei geschlossenen Pfade konvertiert. Bevor der Pfad erweitert wird, konvertiert die Methode alle Bézier-Splines in Polylinien. Sie können für diese Konvertierung optional einen Faktor für die Abflachung angeben oder die Pfadkoordinaten vor der Erweiterung mithilfe eines *Matrix*-Objekts transformieren.

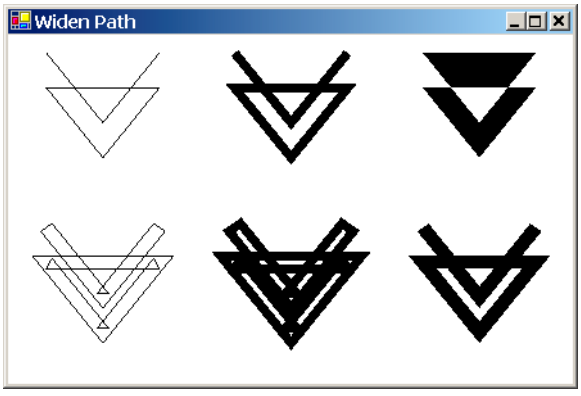
Die *Widen*-Methode führt zuweilen zu etwas eigenartigen Ergebnissen, daher wollen wir uns das Ganze einmal an einem Beispiel anschauen. Das folgende Programm erstellt im Konstruktor einen Pfad, der einen V-förmigen offenen Teilpfad und einen dreieckigen geschlossenen Teilpfad enthält.

WidenPath.cs

```
//----------------------------------------
// WidenPath.cs © 2001 by Charles Petzold
//----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class WidenPath: PrintableForm
{
     GraphicsPath path;
```

```
    public new static void Main()
    {
        Application.Run(new WidenPath());
    }
    public WidenPath()
    {
        Text = "Widen Path";

        path = new GraphicsPath();

            // Offenen Teilpfad erstellen

        path.AddLines(new Point[] { new Point(20, 10),
                                    new Point(50, 50),
                                    new Point(80, 10) });

            // Geschlossenen Teilpfad erstellen

        path.AddPolygon(new Point[] { new Point(20, 30),
                                      new Point(50, 70),
                                      new Point(80, 30) });
    }
    protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
    {

        grfx.ScaleTransform(cx / 300f, cy / 200f);

        for (int i = 0; i < 6; i++)
        {
            GraphicsPath pathClone = (GraphicsPath) path.Clone();
            Matrix       matrix    = new Matrix();
            Pen          penThin   = new Pen(clr, 1);
            Pen          penThick  = new Pen(clr, 5);
            Pen          penWiden  = new Pen(clr, 7.5f);
            Brush        brush     = new SolidBrush(clr);

            matrix.Translate((i % 3) * 100, (i / 3) * 100);

            if (i < 3)
                pathClone.Transform(matrix);
            else
                pathClone.Widen(penWiden, matrix);

            switch (i % 3)
            {
            case 0:  grfx.DrawPath(penThin, pathClone);   break;
            case 1:  grfx.DrawPath(penThick, pathClone);  break;
            case 2:  grfx.FillPath(brush, pathClone);     break;
            }
        }
    }
}
```

Die *DoPage*-Methode kopiert diesen Pfad mithilfe der *Clone*-Methode sechsmal und platziert jede Kopie mit der *Transform*-Methode in einen bestimmten Bildschirmbereich. Anschließend zeichnet sie den Pfad auf sechs verschiedenen Arten. Und so sieht das Ergebnis aus:



In der oberen Reihe wurde der Pfad (von links nach rechts) mit einem 1 Einheit breiten Stift, mit einem 5 Einheiten breiten Stift und gefüllt gezeichnet. Die Objekte der unteren Reihe wurden mit den gleichen Einstellungen gezeichnet, zuvor erfolgte jedoch ein *Widen*-Aufruf mit einem 7,5 Einheiten breiten Stift.

An den beiden linken Objekten lassen sich die Auswirkungen der *Widen*-Methode am deutlichsten erkennen. Der offene, V-förmige Teilpfad wird in einen geschlossenen Teilpfad umgewandelt, der die gleichen Umrisse aufweist, als wäre er mit einem breiten Stift gezeichnet worden. Der geschlossene, dreieckige Teilpfad wurde in zwei Pfade umgewandelt: einer innerhalb und einer außerhalb einer Linie, die entstünde, wenn der Pfad mit einem breiten Stift gezeichnet würde. Die kleinen Schleifen im Inneren der Pfade sehen natürlich etwas merkwürdig aus, rühren aber von dem Algorithmus her, den die *Widen*-Methode einsetzt.

Die beiden in der Mitte abgebildeten Objekte sehen aus wie die linken, wenn sie mit einem breiteren Stift gezeichnet werden.

Der gefüllte Pfad oben rechts weist aufgrund des Standardfüllmodus für den Pfad, nämlich *FillMode.Alternating,* eine ungefüllte Innenfläche auf. Wenn Sie stattdessen den Füllmodus *FillMode.Winding* angeben, werden alle inneren Flächen gefüllt. Am interessantesten ist aber die Figur rechts unten. Hier können Sie erkennen, wie sich *FillPath* auf den erweiterten Pfad auswirkt. Diese Figur sieht aus, als wäre auf den Originalpfad eine *DrawPath*-Methode mit einem breiten Stift angewendet worden.

Mithilfe der *GetBounds*-Methode können Sie das kleinste Rechteck feststellen, in das der Pfad hineinpasst. Dabei können Sie die Auswirkungen einer Matrixtransformation und der Verwendung eines breiten Stifts mit einbeziehen oder auch nicht:

GetBounds-Methoden der *GraphicsPath*-Klasse

```
RectangleF GetBounds()
RectangleF GetBounds(Matrix matrix)
RectangleF GetBounds(Matrix matrix, Pen pen)
```

Keines dieser Argumente hat Auswirkungen auf die im Pfad gespeicherten Koordinaten. Denken Sie daran, dass das berechnete Rechteck die Maximal- und Minimalwerte für die *x*- und *y*-Koordinaten aller Punkte im Pfad darstellt. Enthält der Pfad Bézier-Kurven, so richtet sich das Rechteck nach den Koordinaten der Kontrollpunkte, nicht nach der eigentlichen Kurve. Wenn Sie eine genauere Abmessung der Figur benötigen, sollten Sie zunächst *Flatten* und dann erst *GetBounds* aufrufen.

In Kapitel 7 habe ich die Matrixtransformation als *lineare* Transformation definiert. Durch diesen Aspekt der Linearität ist die Transformation einigen Beschränkungen unterworfen. Parallelogramme beispielsweise werden stets in Parallelogramme transformiert.

Die Klasse *GraphicsPath* stellt in der *Warp*-Methode eine weitere Transformation bereit. Genau wie die *Transform*-Methode verändert auch *Warp* alle Koordinaten eines Pfads. Die *Warp*-Transformation ist jedoch nicht linear. Es handelt sich hier um die einzige nichtlineare Transformation in GDI+.

Um diese Transformation durchführen zu können, müssen Sie vier Quell- und vier Zielkoordinaten angeben. Die Methode ordnet die vier Quellkoordinaten den zugehörigen Zielkoordinaten zu. Die Quellkoordinaten werden als *RectangleF*-Struktur angegeben. Es ist ganz praktisch (aber nicht zwingend erforderlich), das *RectangleF*-Argument auf die von *GetBounds* zurückgegebene *RectangleF*-Struktur zu setzen. Die Zielkoordinaten werden als Array aus *PointF*-Strukturen definiert:

Warp-Methoden der *GraphicsPath*-Klasse

```
void Warp(PointF[] aptfDst, RectangleF rectfSrc)
void Warp(PointF[] aptfDst, RectangleF rectfSrc, Matrix matrix)
void Warp(PointF[] aptfDst, RectangleF rectfSrc, Matrix matrix, WarpMode wm)
void Warp(PointF[] aptfDst, RectangleF rectfSrc, Matrix matrix, WarpMode wm, float fFlatness)
```

Optional können Sie außerdem ein *Matrix*-Objekt und einen Wert für die Abflachung angeben. Die Quellpunkte werden nach folgenden Regeln in Zielpunkte transformiert:

- *aptfDst[0]* gibt das Ziel der oberen linken Ecke des Rechtecks an.
- *aptfDst[1]* gibt das Ziel der oberen rechten Ecke des Rechtecks an.
- *aptfDst[2]* gibt das Ziel der unteren linken Ecke des Rechtecks an.
- *aptfDst[3]* gibt das Ziel der unteren rechten Ecke des Rechtecks an.

Mit einem optionalen Argument können Sie bestimmen, wie die dazwischen liegenden Punkte berechnet werden:

WarpMode-Enumeration

Member	Wert
Perspective	0
Bilinear	1

Das nun folgende Programm PathWarping gibt Ihnen die Möglichkeit, ein bisschen mit der *Warp*-Funktion zu spielen. Der Formularkonstruktor erstellt einen Pfad, der ein Schachbrettmuster mit 8 mal 8 Feldern zeichnet. Das Ziel des Pfads können Sie dann mit der Maus angeben.

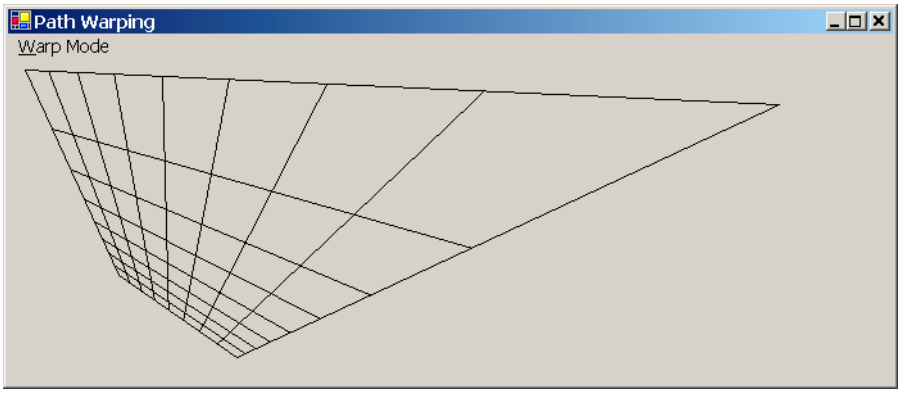
PathWarping.cs

```
//-----------------------------------------
// PathWarping.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
```

```
class PathWarping: Form
{
     MenuItem     miWarpMode;
     GraphicsPath path;
     PointF[]     aptfDest = new PointF[4];

     public static void Main()
     {
          Application.Run(new PathWarping());
     }
     public PathWarping()
     {
          Text = "Path Warping";

               // Menü erstellen

          Menu = new MainMenu();

          Menu.MenuItems.Add("&Warp Mode");
          EventHandler ehClick = new EventHandler(MenuWarpModeOnClick);

          miWarpMode = new MenuItem("&" + (WarpMode)0, ehClick);
          miWarpMode.RadioCheck = true;
          miWarpMode.Checked = true;
          Menu.MenuItems[0].MenuItems.Add(miWarpMode);

          MenuItem mi = new MenuItem("&" + (WarpMode)1, ehClick);
          mi.RadioCheck = true;
          Menu.MenuItems[0].MenuItems.Add(mi);

               // Pfad erstellen

          path = new GraphicsPath();

          for (int i = 0; i <= 8; i++)
          {
               path.StartFigure();
               path.AddLine(0, 100 * i, 800, 100 * i);
               path.StartFigure();
               path.AddLine(100 * i, 0, 100 * i, 800);
          }
               // Punktearray initialisieren

          aptfDest[0] = new Point( 50,  50);
          aptfDest[1] = new Point(200,  50);
          aptfDest[2] = new Point( 50, 200);
          aptfDest[3] = new Point(200, 200);
     }
     void MenuWarpModeOnClick(object obj, EventArgs ea)
     {
          miWarpMode.Checked = false;
          miWarpMode = (MenuItem) obj;
          miWarpMode.Checked = true;

          Invalidate();
     }
     protected override void OnMouseDown(MouseEventArgs mea)
     {
          Point pt;
```

```
        if (mea.Button == MouseButtons.Left)
        {
            if (ModifierKeys == Keys.None)
                pt = Point.Round(aptfDest[0]);
            else if (ModifierKeys == Keys.Shift)
                pt = Point.Round(aptfDest[2]);
            else
                return;
        }
        else if (mea.Button == MouseButtons.Right)
        {
            if (ModifierKeys == Keys.None)
                pt = Point.Round(aptfDest[1]);
            else if (ModifierKeys == Keys.Shift)
                pt = Point.Round(aptfDest[3]);
            else
                return;
        }
        else
            return;

        Cursor.Position = PointToScreen(pt);
    }
    protected override void OnMouseMove(MouseEventArgs mea)
    {
        Point pt = new Point(mea.X, mea.Y);

        if (mea.Button == MouseButtons.Left)
        {
            if (ModifierKeys == Keys.None)
                aptfDest[0] = pt;
            else if (ModifierKeys == Keys.Shift)
                aptfDest[2] = pt;
            else
                return;
        }
        else if (mea.Button == MouseButtons.Right)
        {
            if (ModifierKeys == Keys.None)
                aptfDest[1] = pt;
            else if (ModifierKeys == Keys.Shift)
                aptfDest[3] = pt;
            else
                return;
        }
        else
            return;

        Invalidate();
    }
    protected override void OnPaint(PaintEventArgs pea)
    {
        Graphics     grfx       = pea.Graphics;
        GraphicsPath pathWarped = (GraphicsPath) path.Clone();
        WarpMode     wm         = (WarpMode) miWarpMode.Index;

        pathWarped.Warp(aptfDest, path.GetBounds(), new Matrix(), wm);
        grfx.DrawPath(new Pen(ForeColor), pathWarped);
    }
}
```

Legen Sie mit der linken und der rechten Maustaste die oberen linke und die oberen rechte Zielkoordinate fest. Die untere Zielkoordinate können Sie einstellen, indem Sie beim Klicken mit den Maustasten die Umschalttaste gedrückt halten. Im Menü können Sie zwischen den Modi *Perspective* und *Bilinear* wählen. (Haben Sie bemerkt, wie geschickt die *OnPaint*-Methode die *Index*-Eigenschaft des angeklickten Menüelements in ein Member vom Typ *WarpMode* umwandelt?) So könnte eine perspektivische Verzerrung aussehen:



Ein Pfad bietet eine praktische Möglichkeit, selbst definierte nichtlineare Transformationen zu implementieren. Dazu speichern Sie zunächst die gewünschte Figur in einem Pfad. Anschließend rufen Sie über die *PathPoints*- und *PathTypes*-Eigenschaften die Koordinatenpunkte ab. Verändern Sie diese Punkte beliebig und verwenden Sie dann ein beliebigen *GraphicsPath*-Konstruktor außer dem Standardkonstruktor, um auf der Grundlage der modifizierten Arraywerte einen neuen Pfad zu erstellen. Zwei Beispiele für diese Vorgehensweise finden Sie in Kapitel 19.

Clipping bei Pfaden

Pfade können nicht nur zum Zeichnen und Füllen, sondern auch zur Bestimmung eines Clippingbereichs für das *Graphics*-Objekt eingesetzt werden:

SetClip-Methoden der *Graphics*-Klasse (Auswahl)

```
void SetClip(GraphicsPath path)
void SetClip(GraphicsPath path, CombineMode cm)
```

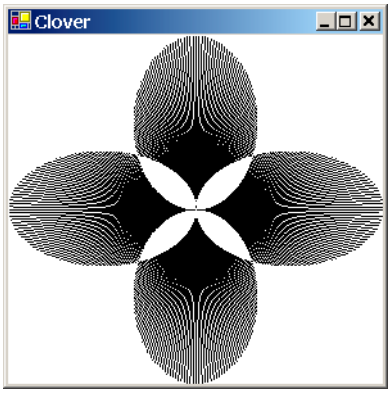
Nehmen wir einmal an, ein Pfad enthalte eine Ellipse. Wenn Sie die erste *SetClip*-Version aufrufen, sind alle nachfolgenden Zeichenoperationen auf diese Ellipse beschränkt. Zur zweiten Version kommen wir gleich. Zunächst wollen wir uns aber auf ein Beispielprogramm stürzen. Das Programm Clover definiert einen Pfad mit vier sich überschneidenden Ellipsen und verwendet diesen als Clippingbereich.

Clover.cs

```
//-----------------------------------
// Clover.cs © 2001 by Charles Petzold
//-----------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class Clover: PrintableForm
{
     public new static void Main()
     {
          Application.Run(new Clover());
     }
     public Clover()
     {
          Text = "Clover";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          GraphicsPath path = new GraphicsPath();

          path.AddEllipse(0,      cy / 3, cx / 2, cy / 3);  // Links
          path.AddEllipse(cx / 2, cy / 3, cx / 2, cy / 3);  // Rechts
          path.AddEllipse(cx / 3, 0,      cx / 3, cy / 2);  // Oben
          path.AddEllipse(cx / 3, cy / 2, cx / 3, cy / 2);  // Unten

          grfx.SetClip(path);
          grfx.TranslateTransform(cx / 2, cy / 2);

          Pen   pen     = new Pen(clr);
          float fRadius = (float) Math.Sqrt(Math.Pow(cx / 2, 2) +
                                            Math.Pow(cy / 2, 2));

          for (float fAngle = 0; fAngle <  (float) Math.PI * 2;
                                 fAngle += (float) Math.PI / 180)
          {
               grfx.DrawLine(pen, 0, 0, fRadius * (float) Math.Cos(fAngle),
                                        -fRadius * (float) Math.Sin(fAngle));
          }
     }
}
```

Das *GraphicsPath*-Objekt wird in der *DoPage*-Methode erstellt. Der Pfad besteht aus vier Ellipsen, die auf der Größe des Clientbereichs oder der Druckerseite basieren. Die *SetClip*-Methode gibt anhand des Pfads den Clippingbereich für das *Graphics*-Objekt an.

Anschließend setzt die *DoPage*-Methode den Ursprung in die Mitte des Zeichenbereichs und zeichnet von dort ausgehend strahlenförmig 360 Linien. Diese Linien werden auf die Ellipsenform beschnitten (clipped):



Ein solches Bild auf andere Weise zu zeichnen, dürfte äußerst schwierig sein. Sie haben sicher gesehen, dass der Bereich, in dem sich die Ellipsen überschneiden, nicht zum Clippingbereich gehört. Das liegt daran, dass wir den Standardfüllmodus *FillMode.Alternate* verwendet haben. Wenn Sie vor dem Aufruf von *SetClip* stattdessen diesen Modus angeben:

```
path.FillMode = FillMode.Winding;
```

werden auch die überlappenden Teile in den Clippingbereich einbezogen.

Das Clipping ist meist sehr zeitaufwendig. Ich habe die *Clover*-Klasse aus *PrintableForm* abgeleitet, sodass Sie das Bild mit einem Klick auf den Clientbereich ausgeben können. Ich muss Sie aber warnen: Die Ausgabe kann eine Stunde und länger dauern. Nun stellt sich natürlich die Frage: Wie wirken sich Seiten- und Welttransformation auf den Clippingbereich aus?

Beim Aufruf von *SetClip* wird angenommen, dass der Pfad in Weltkoordinaten angegeben ist. Die Weltkoordinaten werden in Gerätekoordinaten konvertiert, gerade so, als würden Sie den Pfad zeichnen oder füllen. Der Clippingbereich wird in Gerätekoordinaten gespeichert, und das bleibt auch so. Sie könnten im Clover-Programm beispielsweise nach dem *SetClip*-Aufruf Seiten- und Welttransformation beliebig ändern, und der Zeichenvorgang bliebe doch auf ein und denselben Fensterbereich begrenzt. Ich habe selbst im Programm *TranslateTransform* eingesetzt, ohne dass sich dadurch die Position des Clippingbereichs geändert hätte.

Mit der zweiten oben angeführten *SetClip*-Version können Sie einen vorhandene Clippingbereich mit einem in der *SetClip*-Methode angegebenen zweiten Clippingbereich vereinigen:

CombineMode-Enumeration

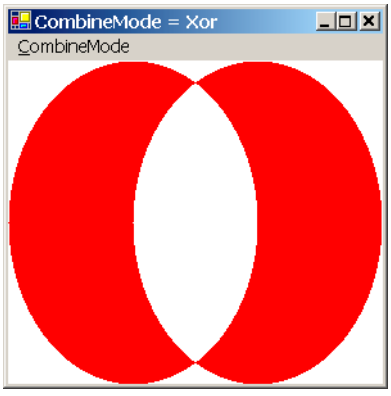
Member	Wert	Beschreibung
Replace	0	Auswahl = Neu
Intersect	1	Auswahl = Neu $\wedge$ Vorhanden
Union	2	Auswahl = Neu $\vee$ Vorhanden
Xor	3	Auswahl = Vereinigungsmenge – Schnittmenge
Exclude	4	Auswahl = Vorhanden – Neu
Complement	5	Auswahl = Neu – Vorhanden

Das folgende Programm erstellt einen Clippingbereich auf der Grundlage der beiden überlappenden Ellipsen. Mithilfe eines Menübefehls können Sie auswählen, welcher *CombineMode*-Wert zur Kombination der beiden Ellipsen verwendet werden soll. Das Programm färbt dann den gesamten Clientbereich ein. Ich habe, genau wie im Programm PathWarping, die Untermenüindizes (die in einem Bereich von 0 bis 5 liegen) als *CombineMode*-Wert verwendet.

ClippingCombinations.cs

```
//-------------------------------------------------
// ClippingCombinations.cs © 2001 by Charles Petzold
//-------------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class ClippingCombinations: PrintableForm
{
     string   strCaption = "CombineMode = ";
     MenuItem miCombineMode;

     public new static void Main()
     {
          Application.Run(new ClippingCombinations());
     }
     public ClippingCombinations()
     {
          Text = strCaption + (CombineMode)0;

          Menu = new MainMenu();
          Menu.MenuItems.Add("&CombineMode");

          EventHandler ehClick = new EventHandler(MenuCombineModeOnClick);

          for (int i = 0; i < 6; i++)
          {
               MenuItem mi   = new MenuItem("&" + (CombineMode)i);
               mi.Click     += ehClick;
               mi.RadioCheck = true;

               Menu.MenuItems[0].MenuItems.Add(mi);
          }
          miCombineMode = Menu.MenuItems[0].MenuItems[0];
          miCombineMode.Checked = true;
     }
     void MenuCombineModeOnClick(object obj, EventArgs ea)
     {
          miCombineMode.Checked = false;
          miCombineMode = (MenuItem) obj;
          miCombineMode.Checked = true;

          Text = strCaption + (CombineMode)miCombineMode.Index;
          Invalidate();
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          GraphicsPath path = new GraphicsPath();
          path.AddEllipse(0, 0, 2 * cx / 3, cy);
          grfx.SetClip(path);

          path.Reset();
          path.AddEllipse(cx / 3, 0, 2 * cx / 3, cy);
          grfx.SetClip(path, (CombineMode)miCombineMode.Index);

          grfx.FillRectangle(Brushes.Red, 0, 0, cx, cy);
     }
}
```

Die Kombination der beiden Ellipsen mit *CombineMode.Xor* führt zu folgendem Ergebnis:



Andere Versionen der *SetClip*-Methode ermöglichen die Angabe des Clippingbereichs als Rechteck (oder eine Kombination des Clippingbereichs mit einem Rechteck) :

SetClip-Methoden der *Graphics*-Klasse (Auswahl)

```
void SetClip(Rectangle rect)
void SetClip(Rectangle rect, CombineMode cm)
void SetClip(RectangleF rectf)
void SetClip(RectangleF rectf, CombineMode cm)
```

Die *Graphics*-Klasse enthält des Weiteren Methoden namens *IntersectClip* und *ExcludeClip,* mit denen der vorhandene Clippingbereich verändert werden kann. Um den Clippingbereich auf den Normalwert (sprich: einen unendlich großen Bereich) zurückzusetzen, rufen Sie folgende Methode auf:

ResetClip-Methode der *Graphics*-Klasse

```
void ResetClip()
```

Clipping bei Bitmaps

Mithilfe von Clipping können Sie auch nicht rechteckige Bereiche von Bitmaps zeichnen. Das folgende Programm lädt ein Bild und definiert im Konstruktor einen Pfad. In der *DoPage*-Methode stellt das Programm anschließend auf der Grundlage dieses Pfads einen Clippingbereich ein und zeichnet die Bitmap.

KeyholeClip.cs

```
//-----------------------------------------
// KeyholeClip.cs © 2001 by Charles Petzold
//-----------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;

class KeyholeClip: PrintableForm
{
     protected Image        image;
     protected GraphicsPath path;
     public new static void Main()
     {
          Application.Run(new KeyholeClip());
     }
     public KeyholeClip()
     {
          Text = "Keyhole Clip";
          image = Image.FromFile("..\\..\\..\\..\\Images and Bitmaps\\Apollo11FullColor.jpg");
          path = new GraphicsPath();
          path.AddArc(80, 0, 80, 80, 45, -270);
          path.AddLine(70, 180, 170, 180);
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {    grfx.SetClip(path);
          grfx.DrawImage(image, 0, 0, image.Width, image.Height);
     }
}
```

Zugegeben, das Ergebnis ist ein bisschen, nun ja, unangemessen (ein Schlüsselloch auf dem Mond?), aber das Verfahren funktioniert:

Ich habe in diesem Programm die Definition des Pfads auf das Bild abgestimmt und angenommen, dass das Bild in Pixeln gezeichnet wird und dass sich die obere linke Ecke am Punkt (0, 0) befindet.

Was aber, wenn das beschnittene Bild in der Mitte des Clientbereichs angezeigt werden soll? Das Zeichnen des Bilds ist dabei gar nicht das Problem, aber wie bekommen Sie den Pfad in die Mitte? Ein Möglichkeit besteht darin, den Pfad in der Größe des Clientbereichs neu zu erstellen. Eine andere Lösung ist die Verschiebung des Pfads mithilfe einer der folgenden Methoden:

TranslateClip-Methoden der *Graphics*-Klasse

```
void TranslateClip(int cx, int cy)
void TranslateClip(float cx, float cy)
```

Das Programm KeyholeClipCentered überschreibt das Programm KeyholeClip und zentriert Clippingbereich und Pfad im Clientbereich.

KeyholeClipCentered.cs

```
//-----------------------------------------------
// KeyholeClipCentered.cs © 2001 by Charles Petzold
//-----------------------------------------------
using System;
using System.Drawing;
using System.Windows.Forms;

class KeyholeClipCentered: KeyholeClip
{
     public new static void Main()
     {
          Application.Run(new KeyholeClipCentered());
     }
     public KeyholeClipCentered()
     {
          Text += " Centered";
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          grfx.SetClip(path);

          RectangleF rectf   = path.GetBounds();
          int        xOffset = (int)((cx - rectf.Width)  / 2 - rectf.X);
          int        yOffset = (int)((cy - rectf.Height) / 2 - rectf.Y);

          grfx.TranslateClip(xOffset, yOffset);
          grfx.DrawImage(image, xOffset, yOffset, image.Width, image.Height);
     }
}
```

Sie erhalten den gleichen Effekt auch, indem Sie eine Bitmap von der Größe des beschnittenen Bilds erstellen und Transparenz verwenden. Das Programm KeyholeBitmap veranschaulicht diese Vorgehensweise.

KeyholeBitmap.cs

```
//-------------------------------------------
// KeyholeBitmap.cs © 2001 by Charles Petzold
//-------------------------------------------
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
using System.Windows.Forms;

class KeyholeBitmap: PrintableForm
{
     Bitmap bitmap;

     public new static void Main()
     {
          Application.Run(new KeyholeBitmap());
     }
     public KeyholeBitmap()
     {
          Text = "Keyhole Bitmap";

               // Bild laden

          Image image = Image.FromFile(
               "..\\..\\..\\..\\Images and Bitmaps\\Apollo11FullColor.jpg");

               // Auswahlpfad erstellen

          GraphicsPath path = new GraphicsPath();
          path.AddArc(80, 0, 80, 80, 45, -270);
          path.AddLine(70, 180, 170, 180);

               // Größe des Auswahlpfads abrufen

          RectangleF rectf = path.GetBounds();

               // Neue, transparente Bitmap erstellen

          bitmap = new Bitmap((int) rectf.Width, (int) rectf.Height,
                              PixelFormat.Format32bppArgb);

               // Basierend auf neuer Bitmap Graphics-Objekt erstellen.

          Graphics grfx = Graphics.FromImage(bitmap);

               // Originalbild beschneiden und auf Bitmap zeichnen

          grfx.SetClip(path);
          grfx.TranslateClip(-rectf.X, -rectf.Y);
          grfx.DrawImage(image, (int) -rectf.X, (int) -rectf.Y,
                                 image.Width, image.Height);
          grfx.Dispose();
     }
     protected override void DoPage(Graphics grfx, Color clr, int cx, int cy)
     {
          grfx.DrawImage(bitmap, (cx - bitmap.Width) / 2,
                                 (cy - bitmap.Height) / 2, bitmap.Width, bitmap.Height);
     }
}
```

Das Laden des Bilds und das Erstellen des Pfads erfolgt auf die gleiche Weise wie im Programm KeyholeClip. Anschließend ruft KeyholeBitmap die Pfadgröße ab und erstellt anhand dieser Größe ein neues *Bitmap*-Objekt. Das Pixelformat der Bitmap ist als *Format32bppArgb* angegeben (das Standardformat) und die Bitmap wird mit Nullwerten (0) initialisiert, ist also vollständig transparent. Das gilt nicht für Objekte, die auf der Bitmap gezeichnet werden, sie sind nicht transparent.

Anschließend ruft der Konstruktor ein *Graphics*-Objekt für die Bitmap ab und legt mit dem Pfad einen Clippingbereich fest. Das Problem dabei ist allerdings, dass die neue Bitmap kleiner ist als die geladene, die Position des Pfads also nicht stimmt. Die *TranslateClip*-Methode verschiebt den Clippingbereich an die richtige Position, *DrawImage* gibt das Bild mit den gleichen Versatzfaktoren auf der neuen Bitmap aus.

Die *DoPage*-Methode zentriert die Bitmap im Clientbereich. Das Programm könnte die neue Bitmap auch speichern.

Bereiche und das Clipping

Historisch betrachtet werden Bereiche von Windows schon erheblich länger unterstützt als Pfade. Bereiche gab es bereits in Windows 1.0 (aus dem Jahr 1985), Pfade dagegen standen erst mit Einführung der 32-Bit-Versionen von Windows zur Verfügung (Windows NT 3.1 im Jahr 1993 und zwei Jahre später Windows 95).

Die Pfade haben die Bereiche in der Windows-Grafikprogrammierung weitgehend verdrängt. Im Grunde spielen Bereiche nur noch beim Thema Clipping eine Rolle. Wenn Sie einen Clippingpfad festlegen, wird dieser Pfad in einen Bereich umgewandelt. Je mehr Sie sich mit dem Thema Clipping beschäftigen, desto gründlicher müssen Sie sich auch mit Bereichen auseinander setzen.

Wie Sie bereits wissen, stellt ein Grafikpfad eine Sammlung aus Linien und Kurven dar. Ein Bereich beschreibt eine Fläche des Ausgabegeräts. Die Konvertierung eines Pfads in einen Bereich ist eine recht einfache Aufgabe. Einer der Konstruktoren der *Region*-Klasse (die im Namespace *System.Drawing* definiert ist) erstellt einen Bereich direkt aus einem Pfad:

Region-Konstruktoren (Auswahl)

```
Region(GraphicsPath path)
```

Damit der Konstruktor funktioniert, werden alle offenen Teilpfade geschlossen. Der Bereich umfasst die Innenflächen aller geschlossenen Teilpfade. Wenn die Teilpfade sich überschneidende Flächen aufweisen, bestimmt der Füllmodus des Pfads, welche Innenflächen in den Bereich aufgenommen werden und welche nicht. Nur eine Methode der *Graphics*-Klasse verwendet zum Zeichnen einen Bereich:

FillRegion-Methode der *Graphics*-Klasse

```
void FillRegion(Brush brush, Region rgn)
```

Wurde der Bereich aus einem Pfad erstellt, entspricht die Methode einem *FillPath*-Aufruf mit dem ursprünglichen Pfad.

Nur eine Version der *SetClip*-Methode setzt Bereiche direkt ein:

SetClip-Methoden der *Graphics*-Klasse (Auswahl)

```
void SetClip(Region rgn, CombineMode cm)
```

Es mag eigenartig anmuten, dass es keine *SetClip*-Version gibt, die ein *Region*-Argument ohne *CombineMode*-Argument besitzt. Das liegt einfach daran, dass die *Clip*-Eigenschaft des *Graphics*-Objekts bereits als *Region* definiert ist. Es folgen drei *Graphics*-Eigenschaften, die sich auf das Clipping beziehen:

Graphics-Eigenschaften (Auswahl)

Typ	Eigenschaft	Zugriff
Region	*Clip*	get/set
RectangleF	*ClipBounds*	get
bool	*IsClipEmpty*	get

Anstatt nun einen Clippingbereich anhand eines Methodenaufrufs aus einem *Region*-Objekt festzulegen:

```
grfx.SetClip(rgn);   // Nicht vorhanden!
```

setzen Sie einfach die Eigenschaft:

```
grfx.Clip = rgn;
```

Die *ClipBounds*-Eigenschaft gibt das kleinste Rechteck an, das den Clippingbereich umfasst, *IsClipEmpty* gibt Aufschluss darüber, ob der Clippingbereich einen nicht vorhandenen Bereich definiert.

Zwei weitere Eigenschaften des *Graphics*-Pfads beziehen sich auf das Clipping:

Graphics-Eigenschaften (Auswahl)

Typ	Eigenschaft	Zugriff
RectangleF	*VisibleClipBounds*	get
bool	*IsVisibleClipEmpty*	get

Bei einem neuen *Graphics*-Objekt gibt die *VisibleClipBounds*-Eigenschaft die Größe der Zeichenfläche an. Hierbei handelt es sich aus Sicht eines Formulars um die Größe des Clientbereichs; der Drucker interpretiert den Wert als Größe des bedruckbaren Seitenbereichs. Die *ClipBounds*-Eigenschaft gibt ein »unendlich großes« Begrenzungsrechteck an. (In Wirklichkeit ist es natürlich nicht unendlich, sondern nur extrem groß.)

Wenn Sie einen Clippingbereich für das *Graphics*-Objekt festlegen, entspricht *VisibleClipBounds* der Schnittmenge der ursprünglichen *VisibleClipBounds* und der *ClipBounds*-Eigenschaft. Befindet sich der Clippingbereich vollständig innerhalb des Anzeigebereichs, sind *VisibleClipBounds* und *ClipBounds* gleich.

Wenn der *IsClipEmpty*-Wert *true* lautet, wird auch *IsVisibleClipEmpty* auf *true* gesetzt. Es kommt vor, dass der *IsClipEmpty*-Wert *true* lautet, sich der Clippingbereich aber außerhalb des Clientbereichs (bzw. des bedruckbaren Bereichs der Druckerseite) befindet. In diesem Fall lautet der *IsVisibleClipEmpty*-Wert trotzdem *true,* da der Clippingbereich vollständig außerhalb des Anzeigebereichs liegt.